БИБЛИОТЕКА ПРОГРАММИСТА

Apress®

Фелипе Гутьеррес

SPRING BOOT 2
ЛУЧШИЕ ПРАКТИКИ ДЛЯ ПРОФЕССИОНАЛОВ

КАК ИЗБАВИТЬСЯ ОТ ЗАБОТ ЭКОСИСТЕМЫ SPRING FRAMEWORK, ИСПОЛЬЗУЯ ВОЗМОЖНОСТИ SPRING BOOT 2

Pro Spring Boot 2

An Authoritative Guide to Building Microservices, Web and Enterprise Applications, and Best Practices

Second Edition

Felipe Gutierrez

Apress

БИБЛИОТЕКА ПРОГРАММИСТА

Фелипе Гутьеррес

SPRING BOOT 2
ЛУЧШИЕ ПРАКТИКИ ДЛЯ ПРОФЕССИОНАЛОВ

КАК ИЗБАВИТЬСЯ ОТ ЗАБОТ ЭКОСИСТЕМЫ SPRING FRAMEWORK, ИСПОЛЬЗУЯ ВОЗМОЖНОСТИ SPRING BOOT 2

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

2020

ББК 32.973.2-018.1
УДК 004.3
Г97

Гутьеррес Фелипе

Г97 Spring Boot 2: лучшие практики для профессионалов. — СПб.: Питер, 2020. — 464 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1587-7

Хотите повысить свою эффективность в разработке корпоративных и облачных Java-приложений?

Увеличьте скорость и простоту разработки микросервисов и сложных приложений, избавившись от забот по конфигурации Spring.

Используйте Spring Boot 2 и такие инструменты фреймворка Spring 5, как WebFlux, Security, Actuator, а также фреймворк Micrometer, предоставляющий новый способ сбора метрик.

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

ББК 32.973.2-018.1
УДК 004.3

Права на издание получены по соглашению с APress Media, LLC, part of Springer Nature. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1484236758 англ.

First published In English under the title Pro Spring Boot 2: An Authoritative Guide to Building Microservices, Web and Enterprise Applications, and Best Practices by Felipe Gutierrez, edition: 2
© Felipe Gutierrez, 2019 *
This edition has been translated and published under licence from APress Media, LLC, part of Springer Nature. APress Media, LLC, part of Springer Nature takes no responsibility and shall not be made liable for the accuracy of the translation.

ISBN 978-5-4461-1587-7
© Перевод на русский язык ООО Издательство «Питер», 2020
© Издание на русском языке, оформление ООО Издательство «Питер», 2020
© Серия «Библиотека программиста», 2020

Краткое содержание

Об авторе ... 16

О научных редакторах ... 17

Благодарности ... 18

Глава 1. Фреймворк Spring 5 ... 19

Глава 2. Введение в Spring Boot ... 45

Глава 3. Внутреннее устройство и возможности Spring Boot ... 59

Глава 4. Создание веб-приложений ... 96

Глава 5. Доступ к данным ... 131

Глава 6. Работа с WebFlux и Reactive Data ... 172

Глава 7. Тестирование ... 203

Глава 8. Безопасность ... 215

Глава 9. Обмен сообщениями ... 257

Глава 10. Spring Boot Actuator ... 301

Глава 11. Создание приложений Spring Integration и Spring Cloud Stream ... 353

Глава 12. Spring Boot в облаке ... 398

Глава 13. Расширение возможностей Spring Boot ... 420

Приложение. Интерфейс командной строки Spring Boot ... 442

Оглавление

Об авторе ... 16

О научных редакторах ... 17

Благодарности ... 18

Глава 1. Фреймворк Spring 5 ... 19

Немного истории ... 19

Принципы и паттерны проектирования ... 20

Фреймворк Spring 5 ... 21

Простое веб-приложение Spring ... 22

Использование Maven для создания проекта ... 23

Добавление зависимостей ... 23

Веб-конфигурация Spring ... 26

Классы ... 33

Запуск приложения ... 38

Использование Java-конфигурации ... 40

Резюме ... 44

Глава 2. Введение в Spring Boot ... 45

Spring Boot ... 45

Spring Boot спешит на помощь ... 47

Spring Boot CLI ... 48

Модель приложения Spring Boot ... 50
Почему Spring Boot? ... 55
Резюме ... 58

Глава 3. Внутреннее устройство и возможности Spring Boot ... 59
Автоматическая конфигурация ... 59
Отключение конкретных автоконфигурационных классов ... 61
Аннотации @EnableAutoConfiguration и @Enable<технология> ... 63
Возможности Spring Boot ... 67
Класс SpringApplication ... 70
Пользовательский баннер ... 71
Класс SpringApplicationBuilder ... 75
Аргументы приложения ... 78
Интерфейсы ApplicationRunner и CommandLineRunner ... 80
Конфигурация приложения ... 82
Примеры использования свойств конфигурации ... 84
Пользовательский префикс для свойств ... 91
Резюме ... 95

Глава 4. Создание веб-приложений ... 96
Spring MVC ... 96
Автоконфигурация Spring Boot MVC ... 97
Spring Boot Web: приложение ToDo ... 99
Приложение ToDo ... 100
Запуск: приложение ToDo ... 111
Тестирование: приложение ToDo ... 112

Spring Boot Web: переопределение настроек по умолчанию 117
Переопределение настроек сервера 117
Формат даты JSON 118
Content-Type: JSON/XML 119
Spring MVC: переопределение настроек по умолчанию 120
Использование другого контейнера приложения 121
Spring Boot Web: клиент 122
Клиентское приложение ToDo 122
Резюме 130

Глава 5. Доступ к данным 131
Базы данных SQL 131
Spring Data 132
Spring JDBC 133
Работа с JDBC в Spring Boot 134
Приложение ToDo с использованием JDBC 135
Spring Data JPA 142
Использование Spring Data JPA со Spring Boot 143
Создание приложения ToDo с использованием Spring Data JPA 144
Spring Data REST 151
Spring Data REST и Spring Boot 152
Приложение ToDo с Spring Data JPA и Spring Data REST 152
Базы данных NoSQL 159
Spring Data MongoDB 159
Использование Spring Data MongoDB со Spring Boot 160
Приложение ToDo с использованием Spring Data MongoDB 162
Приложение ToDo со Spring Data MongoDB REST 165

Spring Data Redis ... 166
Использование Spring Data Redis со Spring Boot ... 166
Приложение ToDo со Spring Data Redis ... 166
Дополнительные возможности по работе с данными с помощью Spring Boot ... 170
Резюме ... 171

Глава 6. Работа с WebFlux и Reactive Data ... 172
Реактивные системы ... 172
Манифест реактивных систем ... 173
Project Reactor ... 174
Создание приложения ToDo с использованием Reactor ... 175
WebFlux ... 183
WebClient ... 184
WebFlux и автоконфигурация Spring Boot ... 185
Использование WebFlux со Spring Boot ... 186
Реактивные данные ... 193
Реактивные потоки данных MongoDB ... 193
Резюме ... 202

Глава 7. Тестирование ... 203
Фреймворк тестирования Spring ... 203
Фреймворк тестирования Spring Boot ... 205
Тестирование конечных точек веб-приложения ... 206
Имитация компонент ... 207
Тестовые срезы Spring Boot ... 208
Резюме ... 214

Глава 8. Безопасность 215
Spring Security 215
Обеспечение безопасности с помощью Spring Boot 216
Приложение ToDo с базовым уровнем безопасности 217
Переопределяем безопасность базового уровня 222
Переопределение используемой по умолчанию страницы входа 224
Пользовательская страница входа 226
Безопасность при использовании JDBC 233
Создание приложения-справочника с использованием средств безопасности JDBC 233
Использование приложения Directory в приложении ToDo 241
Безопасность WebFlux 246
Создание приложения ToDo с OAuth2 247
Создание приложения ToDo в GitHub 250
Резюме 256

Глава 9. Обмен сообщениями 257
Что такое обмен сообщениями 257
Использование JMS со Spring Boot 258
Создание приложения ToDo с использованием JMS 258
Использование паттерна публикации/подписки JMS 268
Удаленный сервер ActiveMQ 269
Использование RabbitMQ со Spring Boot 269
Установка RabbitMQ 270
RabbitMQ/AMQP: точки обмена, привязки и очереди 270
Создание приложения ToDo с помощью RabbitMQ 272
Удаленный сервер RabbitMQ 282

Обмен сообщениями в Redis с помощью Spring Boot 282
Установка Redis 283
Создание приложения ToDo с использованием Redis 283
Удаленный сервер Redis 290
Использование WebSockets со Spring Boot 290
Создание приложения ToDo с использованием WebSockets 290
Резюме 300

Глава 10. Spring Boot Actuator 301
Возможности модуля 302
Создание приложения ToDo с использованием Actuator 302
/actuator 307
/actuator/conditions 308
/actuator/beans 309
/actuator/configprops 310
/actuator/threaddump 310
/actuator/env 311
/actuator/health 312
/actuator/info 313
/actuator/loggers 313
/actuator/loggers/{name} 314
/actuator/metrics 315
/actuator/mappings 316
/actuator/shutdown 317
/actuator/httptrace 319
Изменение идентификатора конечной точки 320
Поддержка CORS в Spring Boot Actuator 320

Изменение пути конечных точек управления приложением 321
Обеспечение безопасности конечных точек 321
Настройка конечных точек 322
Реализация пользовательских конечных точек актуатора 322
Создание приложения ToDo с пользовательскими конечными точками актуатора 323
Конечная точка health Spring Boot Actuator 331
Создание приложения ToDo с пользовательским индикатором состояния приложения 334
Метрики Spring Boot Actuator 339
Создание приложения ToDo с Micrometer: Prometheus и Grafana 339
Получение общей статистики для Spring Boot с помощью Grafana 350
Резюме 352

Глава 11. Создание приложений Spring Integration и Spring Cloud Stream 353
Азбука Spring Integration 354
Программирование Spring Integration 357
Использование XML 362
Использование аннотаций 364
Использование JavaConfig 366
Приложение ToDo с интеграцией чтения файлов 367
Spring Cloud Stream 372
Spring Cloud 372
Spring Cloud Stream 374
«Стартовые пакеты» для приложений Spring Cloud Stream 396
Резюме 397

Глава 12. Spring Boot в облаке 398
Облачная и нативная облачная архитектура 398
Приложения на основе 12 факторов 399
Микросервисы 401
Подготовка приложения ToDo как микросервиса 402
Платформа Pivotal Cloud Foundry 404
PAS: сервис приложений Pivotal 406
Возможности PAS 407
Использование PWS/PAS 408
Cloud Foundry CLI: интерфейс командной строки 410
Вход в PWS/PAS с помощью утилиты CLI 410
Развертывание приложения ToDo в PAS 411
Создание сервисов 414
Убираем за собой 418
Резюме 419

Глава 13. Расширение возможностей Spring Boot 420
Создание spring-boot-starter 420
Модуль todo-client-spring-boot-starter 422
Модуль todo-client-spring-boot-autoconfigure 423
Создание функциональности @Enable* 431
Сервис REST API приложения ToDo 434
Установка и тестирование 437
Проект Task 437
Запуск приложения Task 440
Резюме 441

Приложение. Интерфейс командной строки Spring Boot 442
Spring Boot CLI 442
Команда run 444
Команда test 446
Команда grab 449
Команда jar 450
Команда war 451
Команда install 452
Команда uninstall 453
Команда init 454
Примеры использования команды init 456
Альтернатива команде init 458
Команда shell 458
Команда help 459
Резюме 460

Моей жене Норме Кастанеда

Об авторе

Фелипе Гутьеррес (Felipe Gutierrez) — архитектор ПО, получивший дипломы бакалавра и магистра в области вычислительной техники в Институте технологий и высшего образования города Монтеррей, Мексика. У Гутьерреса более 20 лет опыта в сфере IT, он разрабатывал программы для компаний из множества вертикально интегрированных отраслей, таких как государственное управление, розничная торговля, здравоохранение, образование. В настоящее время он работает в компании Pivotal, кое дело. В настоящее время он работает в компании Pivotal, на PAS и PKS для Cloud Foundry, фреймворке Spring, на приложениях Spring, Groovy и RabbitMQ, помимо прочих акже был архитектором ПО в таких крупных компаниях, как ох и Qualcomm. Гутьеррес — автор книг *Spring Boot Messaging* (Apress, 2017) и *Introducing Spring Framework* (Apress, 2014).

О научных редакторах

Оригинальное издание

Мануэль Жордан Элера (Manuel Jordan Elera) — разработчик-самоучка и исследователь, обожает изучать новые технологии для своих экспериментов и новых их сочетаний. Мануэль получил премии Springy Award Community Champion и Spring Champion 2013. Немногое имеющееся у него свободное время он посвящает чтению Библии и сочинению музыки на гитаре. Мануэль известен под интернет-псевдонимом dr_pompeii. Он осуществлял научную редактуру многих книг, включая *Pro Spring, 4-е издание* (Apress, 2014)[1], *Practical Spring LDAP* (Apress, 2013), *Pro JPA 2, 2-е издание* (Apress, 2013) и *Pro Spring Security* (Apress, 2013).

Русскоязычное издание

Валерий Алексеевич Дмитрущенков работает в IT более 35 лет. Разрабатывал программное обеспечение для множества компаний и отраслей, руководил многими комплексными проектами, включая разработку, внедрение и сопровождение автоматизированных систем. Участвовал в создании крупных проектов для государственных органов, реализованных международными организациями: ООН, USIAD, World Bank в России, Косово, Молдавии и Армении.

[1] *Шефер К., Хо К., Харроп Р.* Spring 4 для профессионалов. 4-е изд. — М.: Вильямс, 2015.

Благодарности

Я хотел бы выразить свою глубочайшую признательность команде издательства Apress: Стиву Энглину (Steve Anglin), принявшему мое предложение издать книгу, Марку Пауэрсу (Mark Powers) — за то, что не давал мне сбиться с пути, и за его терпение, а также остальным работникам Apress, участвовавшим в реализации данного проекта. Спасибо вам всем, что сделали это возможным.

Спасибо моему научному редактору Мануэлю Жордану за развернутость отзывов и затраченные на них усилия, а также всей команде Spring Boot за создание этой замечательной технологии.

Спасибо моим родителям Росио Круз и Фелипе Гутьерресу за их любовь и поддержку; моему брату Эдгару Герардо Гутьерресу и моей невестке Ауристелле Санчес и особенно моим девочкам, поддерживавшим меня, — Норме, Лауре, Наели и Химене, — я люблю вас, девочки. И моему сыну Родриго!

Фелипе Гутьеррес

1 Фреймворк Spring 5

Добро пожаловать в первую главу этой книги, в которой вы познакомитесь с фреймворком Spring, его историей и тем, как он развивался с момента появления. Данная глава предназначена для разработчиков, еще незнакомых со Spring. Если вы опытный разработчик на Spring, можете смело ее пропустить.

Наверное, вы думаете: «Я же хочу изучить Spring Boot. Зачем же мне знать про Spring Framework?» Позвольте поделиться секретом: Spring Boot — это Spring. У Spring Boot просто несколько иной механизм запуска приложений Spring; для полного понимания функционирования Spring Boot необходимо знать больше о фреймворке Spring.

Немного истории

Фреймворк Spring был создан в 2003 году Родом Джонсоном (Rod Johnson), автором руководства *J2EE Development without EJB* (издательство Wrox Publishing, 2004). Фреймворк Spring стал ответом на сложность для понимания на тот момент спецификаций J2EE. Сегодня ситуация улучшилась, но для работы определенных аспектов экосистемы J2EE необходима целая инфраструктура.

Spring можно считать дополнительной технологией для Java EE. Фреймворк Spring включает несколько технологий, таких как API сервлетов (servlet), API веб-сокетов (WebSockets), средства параллельной обработки, API привязки JSON (JSON Binding API), а также проверку достоверности данных (bean validation), JPA, JMS и JTA/JCA.

Фреймворк Spring поддерживает спецификации *внедрения зависимостей* (dependency injection) и *стандартных аннотаций* (common annotation), значительно упрощающие разработку.

В этой главе показано, что фреймворк Spring версий 5.x требует как минимум Java EE версии 7 (Servlet 3.1+ and JPA 2.1). Spring все еще работает с Tomcat 8 и 9, WebSphere 8 и JBoss EAP 7. Кроме того, я покажу вам новое расширение Spring Framework 5 — поддержку реактивности!

Сегодня Spring — один из чаще всего используемых и общепризнанных в сообществе Java фреймворков не только потому, что работает, но и потому, что продолжает внедрять различные новшества, в числе которых, помимо прочих, Spring Boot, Spring Security, Spring Data, Spring Cloud, Spring Batch и Spring Integration.

Принципы и паттерны проектирования

Чтобы разобраться в Spring Boot, необходимо изучить фреймворк; важно понимать не только на что он способен, но и то, каких принципов придерживается. Вот часть принципов фреймворка Spring.

- *Возможность выбора на всех уровнях.* Spring позволяет откладывать принятие проектных решений практически до самого конца. Например, можно менять поставщиков хранилищ в конфигурации без изменения кода. То же самое справедливо и относительно многих других вопросов инфраструктуры и интеграции со сторонними API. Как вам предстоит увидеть, это происходит даже на этапе развертывания приложения в облаке.
- *Приспособление к различным точкам зрения.* Spring приветствует гибкость и не задает никаких жестких путей решения задач. Он поддерживает широкий диапазон потребностей приложений с различными подходами.
- *Обеспечение строгой обратной совместимости.* Развитие Spring всегда старательно направлялось таким образом, чтобы минимизировать число изменений, нарушающих совместимость между версиями. Spring поддерживает тщательно подобранный диапазон версий JDK и сторонних библиотек, упрощая тем самым поддержку использующих его приложений и библиотек.
- *Забота об архитектуре API.* Команда создателей Spring вложила немало труда и размышлений в создание интуитивно понятных API, которые смогут продержаться на протяжении множества версий и лет.

- *Высокие стандарты качества исходного кода.* Spring Framework делает особый акцент на выразительной, актуальной и точной автодокументации (Javadocs). Это один из очень немногих проектов, которые могут по праву заявить о чистой структуре кода без циклических зависимостей между пакетами.

Итак, что же нужно для запуска приложения Spring? Spring работает с простыми Java-объектами в старом стиле (Plain Old Java Objects, POJOs), что сильно упрощает расширение. Spring обеспечивает готовность приложений к промышленной эксплуатации; впрочем, ему нужно немного помочь, добавив конфигурацию, связывающую воедино все зависимости и внедряющую зависимости, необходимые для создания *компонентов* Spring и выполнения приложения (рис. 1.1).

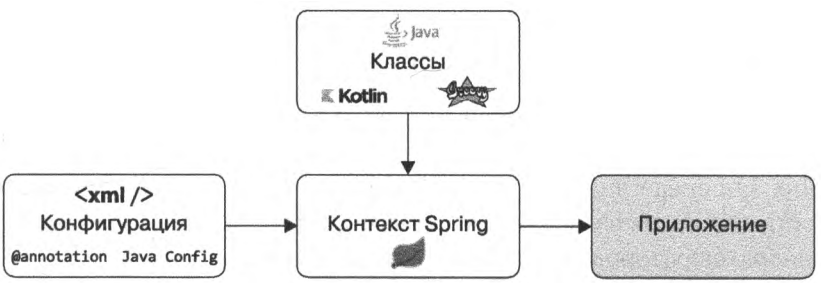


Рис. 1.1. Контекст Spring

Рисунок 1.1 демонстрирует контекст Spring, класс, создающий все компоненты Spring, — с помощью конфигурации со ссылками на все классы, благодаря чему приложение может работать. Больше подробностей вы найдете в следующих разделах, в которых мы создадим полноценное приложение с REST API.

Фреймворк Spring 5

Spring упрощает создание корпоративных приложений Java, поскольку предоставляет все необходимое разработчику для использования языка Java в корпоративной среде. В качестве альтернативных Java-языков на JVM (виртуальной машине Java) он полностью поддерживает Groovy и Kotlin.

Spring Framework 5 требует JDK 8+ и обеспечивает поддержку пакета инструментов Java Development Kit (JDK) версий 9, 10 и 11. Разработчики Spring

гарантируют ту же долгосрочную техническую поддержку версий 11 и 17, что и команда создателей JDK. Эта новая версия, вышедшая в 2017 году, предлагает совершенно новый подход к функциональному программированию на основе реактивных потоков данных (Reactive Streams).

Веб-фреймворк Spring MVC был создан для обслуживания API сервлетов и сервлет-контейнеров. Пока требования к сервисам не повысились, это было нормально, но затем возникла проблема блокировки при каждом запросе; так что для большого количества запросов требовалось другое решение. В результате появился реактивный стек, веб-фреймворк. В версии 5 появился модуль Spring WebFlux с полностью неблокирующим стеком с поддержкой контроля обратного потока данных (Reactive Streams, back pressure), который работает на таких серверах, как Netty, Undertow, а также в контейнерах Servlet 3.1+. Этот неблокирующий стек поддерживает параллельное выполнение при небольшом числе потоков выполнения и способен масштабироваться при изменении аппаратных возможностей.

Модуль WebFlux зависит от другого проекта Spring: *Project Reactor*. Reactor представляет собой рекомендуемую реактивную библиотеку для Spring WebFlux. Он предоставляет типы API Mono и Flux для работы с последовательностями данных из `0..1` и `0..N` элементов соответственно с помощью богатого набора операторов, совпадающего с перечнем операторов API *ReactiveX*. Reactor — библиотека, реализующая интерфейсы Reactive Streams, а потому все ее операторы поддерживают неблокирующий контроль обратного потока данных. Reactor сильно ориентирован на серверный Java и разрабатывался в тесном сотрудничестве со Spring.

Я не стану углубляться в возможности Spring, поскольку лучше показать их на примере простого веб-приложения. Не возражаете? А все эти замечательные возможности WebFlux описываются в отдельной главе[1].

Простое веб-приложение Spring

Начнем с создания веб-приложения Spring — приложения ToDo, предоставляющего REST API, реализующие CRUD-операции (Create, Read, Update and Delete — «создание, чтение, обновление и удаление»). Для создания нового приложения Spring необходимо сначала установить Maven. В следующих главах мы будем использовать либо Maven, либо Cradle.

[1] В главе 6. — *Здесь и далее примеч. пер.*

Использование Maven для создания проекта

Начнем с выполнения следующих команд из Maven для создания проекта ToDo на Spring:

```
$ mvn archetype:generate -DgroupId=com.apress.todo
-DartifactId=todo -Dversion=0.0.1-SNAPSHOT -DinteractiveMode=false
-DarchetypeArtifactId=maven-archetype-webapp
```

Эта команда генерирует основной шаблон и структуру веб-приложения. Обычно она также создает каталоги `webapp` и `resources`, но не каталог `java`, который придется создать вручную.

```
todo
├── pom.xml
└── src
    └── main
        ├── resources
        └── webapp
            ├── WEB-INF
            │   └── web.xml
            └── index.jsp
```

Можете импортировать этот код в свою любимую IDE; это упростит выявление каких-либо проблем.

Добавление зависимостей

Откройте файл `pom.xml` и замените все его содержимое листингом 1.1.

Листинг 1.1. todo/pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
       http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.todo</groupId>
    <artifactId>todo</artifactId>
    <packaging>war</packaging>
    <version>0.0.1-SNAPSHOT</version>
    <name>todo Webapp</name>

    <properties>

        <!-- Универсальные свойства -->
        <java.version>1.8</java.version>
```

```
    <!-- Веб -->
    <jsp.version>2.2</jsp.version>
    <jstl.version>1.2</jstl.version>
    <servlet.version>3.1.0</servlet.version>
    <bootstrap.version>3.3.7</bootstrap.version>
    <jackson.version>2.9.2</jackson.version>
    <webjars.version>0.32</webjars.version>

    <!-- Spring -->
    <spring-framework.version>5.0.3.RELEASE</spring-framework.version>

    <!-- JPA -->
    <spring-data-jpa>1.11.4.RELEASE</spring-data-jpa>
    <hibernate-jpa.version>1.0.0.Final</hibernate-jpa.version>
    <hibernate.version>4.3.11.Final</hibernate.version>

    <!-- Драйверы -->
    <h2.version>1.4.197</h2.version>

    <!-- Журналы -->
    <slf4j.version>1.7.25</slf4j.version>
    <logback.version>1.2.3</logback.version>
</properties>

<dependencies>
    <!-- Spring MVC -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>${spring-framework.version}</version>
    </dependency>

    <!-- Spring Data JPA -->
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-jpa</artifactId>
        <version>${spring-data-jpa}</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate.javax.persistence</groupId>
        <artifactId>hibernate-jpa-2.1-api</artifactId>
        <version>${hibernate-jpa.version}</version>
    </dependency>
    <dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>${hibernate.version}</version>
    </dependency>
```

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>${hibernate.version}</version>
</dependency>

<!-- Журналы -->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jcl-over-slf4j</artifactId>
    <version>${slf4j.version}</version>
</dependency>
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>${logback.version}</version>
</dependency>

<!-- Драйверы -->
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>${h2.version}</version>
    <scope>runtime</scope>
</dependency>

<!-- Веб-зависимости Java EE-->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>${jstl.version}</version>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>${servlet.version}</version>
    <scope>provided</scope>
</dependency>
<dependency>
    <groupId>javax.servlet.jsp</groupId>
    <artifactId>jsp-api</artifactId>
    <version>${jsp.version}</version>
    <scope>provided</scope>
</dependency>

<!-- Веб UI -->
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>webjars-locator</artifactId>
```

```
            <version>${webjars.version}</version>
        </dependency>

        <dependency>
            <groupId>org.webjars</groupId>
            <artifactId>bootstrap</artifactId>
            <version>${bootstrap.version}</version>
        </dependency>

        <!-- Веб - JSON/XML ответ -->
        <dependency>
            <groupId>com.fasterxml.jackson.core</groupId>
            <artifactId>jackson-databind</artifactId>
            <version>${jackson.version}</version>
        </dependency>
        <dependency>
            <groupId>com.fasterxml.jackson.datatype</groupId>
            <artifactId>jackson-datatype-joda</artifactId>
            <version>${jackson.version}</version>
        </dependency>

        <dependency>
            <groupId>com.fasterxml.jackson.dataformat</groupId>
            <artifactId>jackson-dataformat-xml</artifactId>
            <version>${jackson.version}</version>
        </dependency>
    </dependencies>

    <build>
        <finalName>todo</finalName>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

Веб-конфигурация Spring

Начнем с конфигурации Spring. Spring требует, чтобы разработчик указал, где располагаются классы и как они взаимодействуют друг с другом, а также описал кое-какую дополнительную конфигурацию веб-приложений.

Приступим к редактированию файла `web.xml`, показанного в листинге 1.2.

Листинг 1.2. todo/src/main/webapp/WEB-INF/web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" version="2.5">
    <display-name>ToDo Web Application</display-name>
    <servlet>
        <servlet-name>dispatcherServlet</servlet-name>
         <servlet-class>org.springframework.web.servlet.DispatcherServlet
         </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcherServlet</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

Необходимо задать конфигурацию класса `DispatcherServlet` — основной точки входа любого приложения Spring. Этот класс связывает все приложение, базируясь на конфигурации контекста. Как видите, данная конфигурация очень проста.

Создадим файл `dispatcherServlet-servlet.xml` для описания конфигурации контекста Spring. Существует следующее соглашение об именах: если в файле `web.xml` сервлет назван `todo`, то файл контекста Spring должен называться `todo-servlet.xml`. В данном случае сервлет получил название `dispatcherServlet`, так что Spring будет искать файл `dispatcherServlet-servlet.xml` (листинг 1.3).

Листинг 1.3. Файл todo/src/main/webapp/WEB-INF/dispatcherServlet-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xmlns:jpa="http://www.springframework.org/schema/data/jpa"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/jdbc
             http://www.springframework.org/schema/jdbc/spring-jdbc-4.3.xsd
```

```
        http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-4.3.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/
            spring-context-4.3.xsd
        http://www.springframework.org/schema/data/jpa
        http://www.springframework.org/schema/data/
            jpa/spring-jpa-1.8.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-4.3.xsd">

<context:component-scan base-package="com.apress.todo" />

<mvc:annotation-driven>
    <mvc:message-converters>
        <bean class="org.springframework.http.converter.json.
            MappingJackson2HttpMessageConverter">
            <property name="objectMapper" ref="jsonMapper"/>
        </bean>
        <bean class="org.springframework.http.converter.xml.
                    MappingJackson2XmlHttpMessageConverter">
           <property name="objectMapper" ref="xmlMapper"/>
        </bean>

    </mvc:message-converters>
</mvc:annotation-driven>

<bean id="jsonMapper" class="org.springframework.http.converter.json.
                            Jackson2ObjectMapperFactoryBean">
    <property name="simpleDateFormat" value="yyyy-MM-dd HH:mm:ss" />
</bean>
<bean id="xmlMapper" parent="jsonMapper">
    <property name="createXmlMapper" value="true"/>
</bean>

<mvc:resources mapping="/webjars/**"
               location="classpath:META-INF/resources/webjars/" />

<jpa:repositories base-package="com.apress.todo.repository" />

<jdbc:embedded-database id="dataSource" type="H2">
    <jdbc:script location="classpath:META-INF/sql/schema.sql" />
    <jdbc:script location="classpath:META-INF/sql/data.sql" />
</jdbc:embedded-database>
<bean id="jpaVendorAdapter"
   class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
   <property name="showSql" value="true" />
</bean>
```

```
    <bean id="entityManagerFactory"
          class="org.springframework.orm.jpa.
                 LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter" ref="jpaVendorAdapter"/>
    </bean>

    <bean id="transactionManager"
          class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory"/>
    </bean>

    <tx:annotation-driven transaction-manager="transactionManager" />

    <bean class="org.springframework.web.servlet.view.
                 InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/views/" />
        <property name="suffix" value=".jsp" />
    </bean>

    <bean id="h2WebServer" class="org.h2.tools.Server"
          factory-method="createWebServer"
          init-method="start" destroy-method="stop">
         <constructor-arg value="-web,-webAllowOthers,-webDaemon,
                                 -webPort,8082" />
    </bean>

</beans>
```

В листинге 1.3 приведена веб-конфигурация Spring. Рассмотрим все используемые в ней пространства имен XML. Благодаря этому при применении IDE с автодополнением кода у вас будут все компоненты и их атрибуты для каждой точки входа. Проанализируем их.

- `<context:component-scan/>`. Этот тег сообщает *контейнеру Spring* о необходимости просмотреть все классы в поиске аннотаций, включая `@Service` и `@Configuration`. Это помогает Spring в связывании воедино всех компонентов Spring, необходимых для работы приложения. В данном случае будет выполнен просмотр всех снабженных аннотациями классов на уровне пакета `com.apress.todo.*` и всех его подпакетов.
- `<mvc:annotation-driven/>`. Этот тег указывает контейнеру Spring, что речь идет о веб-приложении, так что нужно найти все классы с аннотациями `@Controller` и `@RestController`, а также их методы с аннотацией `@RequestMapping` или другими аннотациями Spring MVC, чтобы создать нужные MVC-компоненты для приема запросов от пользователя.

- `<mvc:message-converters/>`. Этот тег сообщает компонентам MVC, что следует использовать для преобразования сообщений в случае запроса. Например, при запросе с HTTP-заголовком `Accept: application/xml` Spring отвечает в формате XML, аналогично происходит в случае `application/json`.
- Компоненты `jsonMapper` и `xmlMapper`. Эти классы представляют собой компоненты Spring, которые помогают форматировать данные и создавать подходящее средство отображения (mapper).
- `<mvc:resources/>`. Этот тег сообщает Spring MVC, какие ресурсы использовать и где их искать. В данном случае наше приложение задействует набор библиотек *WebJars* (описанный в файле `pom.xml`).
- `<jpa:repositories/>`. Этот тег указывает контейнеру Spring и модулю Spring Data, где расположены интерфейсы, расширяющие интерфейс `CrudRepository`. В данном случае искать их нужно на уровне пакета `com.apress.todo.repository.*`.
- `<jdbc:embedded-database/>`. Поскольку наше приложение использует JPA и драйвер H2 для размещаемой в оперативной памяти базы данных, этот тег лишь описывает применение утилиты, способной выполнять при запуске приложения SQL-скрипт; в данном случае он создает таблицу `todo` и вставляет в нее несколько записей.
- Компонент `jpaVendorAdapter`. Объявление этого компонента необходимо для использования реализации JPA, в данном случае Hibernate (указанной в файле `pom.xml` зависимости). Другими словами, фреймворк Hibernate используется в качестве реализации API постоянного хранения объектов Java (Java Persistence API, JPA).
- Компонент `EntityManagerFactory`. Для каждой из реализаций JPA необходимо создать менеджер сущностей для хранения всех сеансов и выполнения всех SQL-запросов по поручению приложения.
- Компонент `TransactionManager`. Нашему приложению необходима транзакционность, нам ведь не нужны дубликаты или испорченные данные, правда? Необходима полная согласованность с требованиями ACID (атомарность, согласованность, изоляция и сохраняемость), так что без транзакций не обойтись.
- `<tx:annotation-driven/>`. Эта аннотация служит для настройки транзакций на основе предыдущих объявлений.
- Компонент `viewResolver`. Необходимо указать, какой тип механизма представлений будет использовать наше веб-приложение, поскольку вариантов очень много, например Java Server Faces, JSP и т. д.

- Компонент `h2WebServer`. Благодаря настройке с помощью этого компонента, механизма H2, к нему можно будет обращаться из приложения.

Как видите, здесь нужны определенные знания об устройстве приложений Spring. Если вы хотите разобраться в этом более детально, рекомендую заглянуть в несколько книг от издательства Apress, включая I. Cosmina *Pro Spring 5*[1].

Я хочу показать, что нужно сделать для работы простого REST API; и поверьте, если вам кажется, что это слишком сложно, то попробуйте сделать то же самое, включая все возможности нашего приложения (MVC, JPA, SQL-инициализацию, JSP, транзакции), с помощью Java EE.

Взглянем на выполняемые при запуске SQL-скрипты. Создайте следующие два файла в каталоге `resources/META-INF/sql` (листинги 1.4 и 1.5).

Листинг 1.4. Файл todo/src/main/resources/META-INF/sql/schema.sql

```
create table todo (
  id varchar(36) not null,
  description varchar(255) not null,
  created timestamp,
  modified timestamp,
  completed boolean,
  primary key (id)
);
```

Как видите, создается очень простая SQL-таблица.

Листинг 1.5. Файл todo/src/main/resources/META-INF/sql/data.sql

```
insert into todo values ('7fd921cfd2b64dc7b995633e8209f385','Buy
  Milk','2018-09-23 15:00:01','2018-09-23 15:00:01',false);
insert into todo values ('5820a4c2abe74f409da89217bf905a0c','Read a
  Book','2018-09-02 16:00:01','2018-09-02 16:00:01',false);
insert into todo values ('a44b6db26aef49e39d1b68622f55c347','Go to Spring
  One 2018','2018-09-18 12:00:00','2018-09-18 12:00:00',false);
```

И конечно, несколько операторов вставки в таблицу `todo` значений.

Важно понимать, что JPA требует отдельного файла конфигурации средства постоянного хранения данных, где можно указать, например, какие управляемые контейнером классы входят в блок постоянного хранения данных, как и каким таблицам базы данных они соответствуют, подключения к источникам данных и т. д. Поэтому необходимо его создать. Создайте файл `persistence.xml` в каталоге `resources/META-INF/` (листинг 1.6).

[1] *Козмина Ю. и др.* Spring 5 для профессионалов. — М.: Диалектика-Вильямс, 2019.

Листинг 1.6. Файл todo/src/main/resources/META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
                 http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
                 version="1.0">
    <persistence-unit name="toDo">
        <description>My Persistence Unit</description>
    </persistence-unit>
</persistence>
```

Объявлять здесь соответствия классов или подключения не обязательно, поскольку эту задачу берет на себя модуль Spring Data; необходимо только объявить название `persistence-unit`.

Приложению обязательно необходимо журналирование, не только для отладки, но и для того, чтобы видеть, что происходит внутри приложения. Создайте файл `logback.xml` в каталоге `resources` (листинг 1.7).

Листинг 1.7. Файл todo/src/main/resources/logback.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration xmlns="http://ch.qos.logback/xml/ns/logback"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://ch.qos.logback/xml/ns/logback
                   http://ch.qos.logback/xml/ns/logback/logback.xsd">

    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <layout class="ch.qos.logback.classic.PatternLayout">
            <Pattern>
                %d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n
            </Pattern>
        </layout>
    </appender>

    <logger name="org.springframework" level="info" additivity="false">
        <appender-ref ref="STDOUT" />
    </logger>

    <logger name="org.springframework.jdbc" level="debug" additivity="false">
        <appender-ref ref="STDOUT" />
    </logger>

    <logger name="com.apress.todo" level="debug" additivity="false">
        <appender-ref ref="STDOUT" />
    </logger>
```

```
    <root level="error">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

Опять же ничего хитрого. Обратите внимание, что уровень журналирования для `com.apress.todo` установлен в значение `DEBUG`.

Классы

Пришло время написать собственно код для REST API приложения ToDo. Начнем с создания модели предметной области: классов предметной области `ToDo`. Создайте следующие классы в каталоге `src/main/java`.

Учтите, что утилита Maven этой структуры каталогов не создает, ее необходимо создавать вручную (листинг 1.8).

Листинг 1.8. Файл todo/src/main/java/com/apress/todo/domain/ToDo.java

```
package com.apress.todo.domain;

import org.hibernate.annotations.GenericGenerator;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import java.sql.Timestamp;

@Entity
public class ToDo {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    private String description;

    private Timestamp created;
    private Timestamp modified;

    private boolean completed;

    public ToDo() {
    }
```

```
    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public String getDescription() {
        return description;
    }

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

    public Timestamp getCreated() {
        return created;
    }

    public void setCreated(Timestamp created) {
        this.created = created;
     }

    public Timestamp getModified() {
        return modified;
    }

    public void setModified(Timestamp modified) {
        this.modified = modified;
    }

    public boolean isCompleted() {
        return completed;
    }

    public void setCompleted(boolean completed) {
        this.completed = completed;
    }
}
```

Как видите, это обычный класс Java, но поскольку приложение предусматривает постоянное хранение данных (в нашем случае списка дел), то необходимо снабдить вышеупомянутый класс аннотацией `@Entity` и объявить *первичный ключ* (primary key) с аннотацией `@Id`. Этот класс также использует еще одну дополнительную аннотацию для генерации 36-символьного случайного GUID для первичного ключа.

Создадим репозиторий, обеспечивающий все CRUD-операции. В нашем случае приложение использует возможности модуля Spring Data, который скрывает весь стереотипный код для соответствия классов и таблиц и поддерживает сеансы и даже выполняет транзакции. Spring Data реализует весь набор CRUD; другими словами, вам не нужно заботиться о сохранении, обновлении, удалении и поиске записей.

Создайте интерфейс `ToDoRepository`, расширяющий интерфейс `CrudRepository` (листинг 1.9).

Листинг 1.9. Файл todo/src/main/java/com/apress/todo/repository/ToDoRepository.java

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends CrudRepository<ToDo,String> {
}
```

В листинге 1.9 показан интерфейс `ToDoRepository`, расширяющий параметризованный интерфейс `CrudRepository<T,K>`. В качестве параметров `CrudRepository` требуются класс предметной области и тип первичного ключа; в данном случае классом предметной области служит класс `ToDo`, а типом первичного ключа — `String` (снабженный аннотацией `@Id`).

В XML-конфигурации использовался тег `<jpa:repositories/>`, который указывает на пакет `ToDoRepository`, означая, что Spring Data сохраняет запись и связывает воедино все относящееся к расширяющим `CrudRepository` интерфейсам.

Теперь создадим веб-контроллер, который будет принимать запросы от пользователей. Создайте класс `ToDoController` (листинг 1.10).

Листинг 1.10. Файл todo/src/main/java/com/apress/todo/controller/ToDoController.java

```
package com.apress.todo.controller;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
```

```
import org.springframework.http.ResponseEntity;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;

import javax.servlet.http.HttpServletRequest;

@Controller
@RequestMapping("/")
public class ToDoController {

    private ToDoRepository repository;

    @Autowired
    public ToDoController(ToDoRepository repository) {
        this.repository = repository;
    }

    @GetMapping
    public ModelAndView index(ModelAndView modelAndView,
           HttpServletRequest request) {
        modelAndView.setViewName("index");
        return modelAndView;
    }

     @RequestMapping(value = "/toDos", method = { RequestMethod.GET },
     produces = {
             MediaType.APPLICATION_JSON_UTF8_VALUE,
             MediaType.APPLICATION_XML_VALUE,
             MediaType.TEXT_XML_VALUE})
     public ResponseEntity<Iterable<ToDo>> getToDos(@RequestHeader
     HttpHeaders headers) {
         return new ResponseEntity<Iterable<ToDo>>(this.repository.findAll(),
                 headers, HttpStatus.OK);
    }
}
```

В листинге 1.10 показан веб-контроллер. Посмотрите на этот код внимательно. Для описания всех модулей и функциональных возможностей Spring MVC нам понадобилась целая книга.

Важный нюанс: класс снабжен аннотацией `@Controller`. Помните тег `<mv:annotation-driven/>`? Он находит все помеченные аннотацией `@Controller` классы и регистрирует контроллеры со всеми снабженными аннотациями `@GetMapping`, `@RequestMapping` и `@PostMapping` методами для получения запросов в соответствии с описанными путями. В данном случае описаны только пути `/` и `/toDos`.

В качестве параметра конструктор этого класса принимает объект `ToDoRepository`, внедряемый контейнером Spring благодаря аннотации `@Autowired`. Эту аннотацию можно опустить при использовании Spring версии 4.3; в ней по умолчанию контейнер Spring определяет нужные конструктору зависимости и внедряет их автоматически. Это все равно что сказать контейнеру Spring: «Эй, контейнер Spring, я собираюсь использовать компонент `ToDoRepository`, внедри его, пожалуйста». Именно так Spring применяет внедрение зависимостей (существует также внедрение методов, внедрение полей и внедрение сеттеров).

Аннотация `@GetMapping` (`@RequestMapping` по умолчанию делает то же самое) задает обработчик для пути `/` и имя представления; в данном случае она возвращает соответствующее JSP-странице `WEB-INF/view/index.jsp` имя index. Аннотация `@RequestMapping` — еще один способ сделать то же самое (что и `@GetMapping`), но она объявляет путь `/toDos`. Получаемый от этого метода ответ зависит от типа заголовка, отправленного инициатором запроса, например `application/json` или `application/xml`. В качестве ответа используется объект `ResponseEntity`; для вызова метода `findAll` (возвращающего все запланированные дела (ToDo)) используется экземпляр репозитория, поскольку в конфигурации средств отображения для типов содержимого JSON и XML объявлено, что это преобразование берет на себя движок.

Опять же не торопитесь и внимательно проанализируйте происходящее. После запуска приложения вы сможете поэкспериментировать со всеми этими аннотациями.

Создадим представление, являющее собой JSP-страницу, вызываемую при обращении к пути `/`. Создайте файл `index.jsp` в каталоге `WEB-INF/views` (листинг 1.11).

Листинг 1.11. Файл todo/src/main/webapp/WEB-INF/views/index.jsp

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!doctype html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Simple Directory Web App</title>
    <link rel="stylesheet" type="text/css"
          href="webjars/bootstrap/3.3.7/css/bootstrap.min.css">
    <link rel="stylesheet" type="text/css"
          href="webjars/bootstrap/3.3.7/css/bootstrap-theme.min.css">
</head>
```

```
<body>
<div class="container theme-showcase" role="main">
    <div class="jumbotron">
        <h1>ToDo Application</h1>
        <p>A simple Rest API Spring MVC application</p>
    </div>
    <div class="page-header">
        <h1>API</h1>
        <a href="toDos">Current ToDos</a>
    </div>
</div>
</body>
</html>
```

Единственное, на что стоит здесь обратить внимание, — использование ресурсов, например набора библиотек WebJars. Приложение задействует CSS Bootstrap. Но откуда эти ресурсы берутся? Во-первых, в файле `pom.xml` объявлена зависимость `org.webjars:bootstrap`. Во-вторых, в конфигурации использовался тег `<mvc:resources/>`, указывающий, где эти ресурсы искать.

Запуск приложения

Мы завершили написание конфигурации и необходимого для работы приложения кода. Настало время сервера приложений. Для запуска приложения следуйте инструкции.

1. Откройте терминал и перейдите в корневой каталог проекта (`todo/`). Выполните следующую команду Maven:

   ```
   $ mvn clean package
   ```

   Эта команда упакует наше приложение в WAR-файл (веб-архив), готовый для развертывания на сервере приложений. Этот файл будет располагаться в каталоге `target/` и называться `todo.war`.

2. Скачайте сервер приложений Tomcat (для работы этого приложения не нужны тяжеловесные серверы приложений; вполне достаточно облегченного Tomcat). Скачать его можно по адресу: https://tomcat.apache.org/download-90.cgi.

3. Распакуйте его и установите в любой каталог.

4. Скопируйте файл `target/todo.war` в каталог `<tomcat-installation>/webapps/`.

5. Запустите Tomcat. Запустите браузер и перейдите по адресу http://localhost:8080/todo (рис. 1.2).

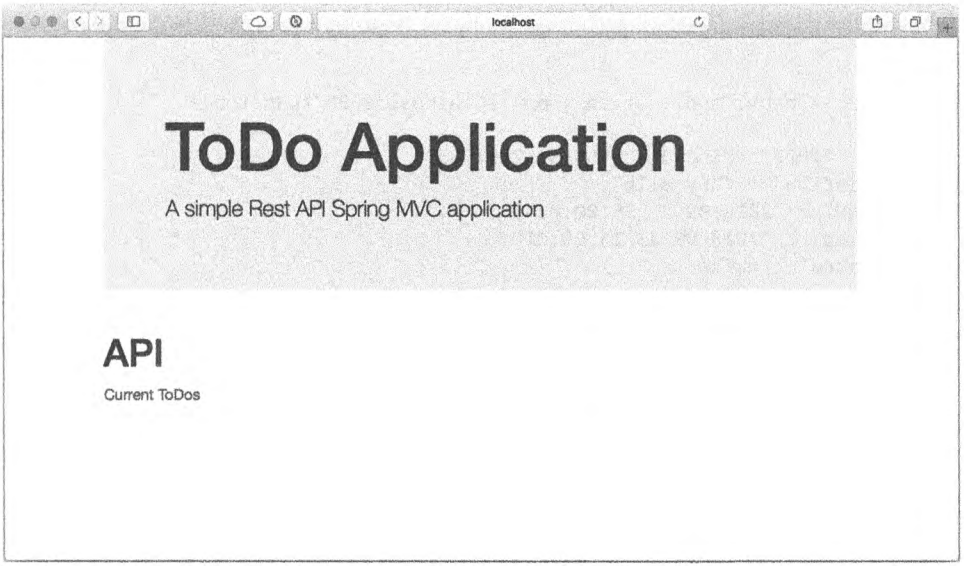


Рис. 1.2. http://localhost:8080/todo/

Если вы щелкнете на ссылке, то получите XML-ответ (рис. 1.3).

Рис. 1.3. http://localhost:8080/todo/toDos

Как получить JSON-ответ? Откройте терминал и выполните следующую команду.

```
$ curl -H "Accept: application/json" localhost:8080/todo/toDos
[ {
  "id" : "7fd921cfd2b64dc7b995633e8209f385",
  "description" : "Buy Milk",
  "created" : "2018-09-23 15:00:01",
  "modified" : "2018-09-23 15:00:01",
  "completed" : false
}, {
  "id" : "5820a4c2abe74f409da89217bf905a0c",
  "description" : "Read a Book",
  "created" : "2018-09-02 16:00:01",
  "modified" : "2018-09-02 16:00:01",
  "completed" : false
}, {
  "id" : "a44b6db26aef49e39d1b68622f55c347",
  "description" : "Go to Spring One 2018",
  "created" : "2018-09-18 12:00:00",
  "modified" : "2018-09-18 12:00:00",
  "completed" : false
} ]
```

Можете проверить с `application/xml` — вы увидите тот же результат, что и в браузере. Поздравляю! Вы только что создали свое первое приложение MVC Spring с REST API.

Использование Java-конфигурации

Возможно, XML представляется вам слишком «многословным» способом описания конфигурации. Что ж, иногда так и есть, но в Spring имеется и другой способ задания настроек контейнера Spring — с помощью аннотаций и классов конфигурации на языке Java.

Если хотите попробовать этот способ, создайте класс `ToDoConfig` и вставьте в него код, приведенный в листинге 1.12.

Листинг 1.12. Файл todo/src/main/java/com/apress/todo/config/ToDoConfig.java

```
package com.apress.todo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.http.converter.HttpMessageConverter;
```

```
import org.springframework.http.converter.json.Jackson2ObjectMapperBuilder;
import org.springframework.http.converter.json.
MappingJackson2HttpMessageConverter;
import org.springframework.http.converter.xml.
    MappingJackson2XmlHttpMessageConverter;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.
    ResourceHandlerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.resource.WebJarsResourceResolver;
import org.springframework.web.servlet.view.InternalResourceViewResolver;

import javax.sql.DataSource;
import java.text.SimpleDateFormat;
import java.util.List;

@Configuration
@EnableJpaRepositories(basePackages="com.apress.todo.repository")
@EnableTransactionManagement
@EnableWebMvc
public class ToDoConfig implements WebMvcConfigurer {

    @Override
    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry
          .addResourceHandler("/webjars/**")
          .addResourceLocations("classpath:/META-INF/resources/webjars/","/
              resources/","/webjars/")
              .resourceChain(true).addResolver(new WebJarsResourceResolver());
    }

    @Override
    public void configureMessageConverters(List<HttpMessageConverter<?>>
        converters) {
          Jackson2ObjectMapperBuilder builder = new
          Jackson2ObjectMapperBuilder();
          builder.indentOutput(true).dateFormat(new
          SimpleDateFormat("yyyy-MM-dd HH:mm:ss"));
          converters.add(new MappingJackson2HttpMessageConverter(
          builder.build()));
          converters.add(new MappingJackson2XmlHttpMessageConverter(
          builder.createXmlMapper(true).build()));
    }
```

```
    @Bean
    public InternalResourceViewResolver jspViewResolver() {
        InternalResourceViewResolver bean = new InternalResourceViewResolver();
        bean.setPrefix("/WEB-INF/views/");
        bean.setSuffix(".jsp");
        return bean;
    }

    @Bean
    public DataSource dataSource() {
        EmbeddedDatabaseBuilder builder =
            new EmbeddedDatabaseBuilder();
        return builder.setType(EmbeddedDatabaseType.H2).addScript("META-INF/
            sql/schema.sql").addScript("META-INF/sql/data.sql").build();
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        HibernateJpaVendorAdapter vendorAdapter =
            new HibernateJpaVendorAdapter();
        vendorAdapter.setShowSql(true);

        LocalContainerEntityManagerFactoryBean factory =
            new LocalContainerEntityManagerFactoryBean();
        factory.setJpaVendorAdapter(vendorAdapter);
        factory.setDataSource(dataSource());
        return factory;
    }
    @Bean
    public PlatformTransactionManager transactionManager() {
        JpaTransactionManager txManager = new JpaTransactionManager();
        txManager.setEntityManagerFactory(entityManagerFactory().
            getNativeEntityManagerFactory());
        return txManager;
    }
}
```

Листинг 1.12, по сути, то же самое, что XML-конфигурация, только теперь используется класс конфигурации на языке Java, в котором мы объявляем компоненты Spring программным образом, переопределяя часть веб-конфигурации.

Если хотите проверить его на деле, необходимо кое-что сделать. Откройте файл `dispatcherServlet-servlet.xml`, который должен выглядеть так:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
```

```
        xsi:schemaLocation="
          http://www.springframework.org/schema/beans http://www.
              springframework.org/schema/beans/spring-beans.xsd
          http://www.springframework.org/schema/context http://www.
              springframework.org/schema/context/spring-context-4.3.xsd">

    <context:component-scan base-package="com.apress.todo" />

</beans>
```

В конце этого файла необходимо сообщить Spring, где искать класс, снабженный аннотацией `@Configuration` (в качестве альтернативы можно использовать класс `WebApplicationInitializer`); когда Spring найдет его, тогда сможет связать приложение воедино на основе объявлений в классе Java-конфигурации.

Не забудьте удалить и заново упаковать свое приложение с помощью команды `mvn clean package` для повторной генерации WAR-файла. Можете запустить его. Вы должны получить такой же результат, что и при использовании XML-конфигурации.

Итак, как вам фреймворк Spring? Да, конечно, нужно разобраться, что происходит. Необходимо понять, как выглядит жизненный цикл компонентов Spring и как применяется внедрение зависимостей. Важно также иметь хотя бы небольшое представление об AOP (аспектно-ориентированном программировании), поскольку в том числе и благодаря ему связывается воедино работающее приложение Spring.

Не слишком ли много всего? Что ж, попробуйте создать точно такое же приложение с помощью обычного J2EE — и усилий на это придется потратить еще больше. Помните, оно не только предоставляет REST API, но и работает с базой данных, транзакциями, конвертерами сообщений, распознаванием представлений и многим другим; именно поэтому веб-приложения легче создавать с помощью Spring.

Но знаете что? Spring Boot предоставляет все шаблоны конфигурации, что еще более ускоряет разработку корпоративных приложений Spring!

ПРИМЕЧАНИЕ

Исходный код для этой книги можно найти на сайте Apress или в GitHub по адресу https://github.com/Apress/pro-spring-boot-2.

Резюме

Говорить о фреймворке Spring и роли Spring Boot можно очень долго. Одной главы, конечно, недостаточно. Так что, если хотите узнать больше, рекомендую заглянуть в документацию Spring по адресу https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/.

В следующей главе мы приступим к работе со Spring Boot и увидим, насколько легко можно создать то же самое приложение, но «а-ля Boot».

2 Введение в Spring Boot

В предыдущей главе я показал вам, что такое фреймворк Spring, часть основных его возможностей (в частности, реализацию паттерна проектирования «Внедрение зависимостей»), а также продемонстрировал его использование на примере создания простого веб-приложения и его развертывания на сервере Tomcat. Мы также проследили все шаги создания приложения Spring (например, параметры конфигурации с добавлением различных XML-файлов и способ запуска приложения).

В этой главе я покажу вам, что такое Spring Boot — его основные компоненты, как создавать, запускать и развертывать приложения Spring с его помощью. Spring Boot упрощает создание приложений Spring. Подробности описаны в оставшейся части данной книги, здесь вас ждет только небольшое введение в технологию Spring Boot.

Spring Boot

Можно сказать, что Spring Boot — следующий этап эволюции фреймворка Spring, но не поймите меня превратно: Spring Boot — вовсе не замена фреймворку Spring, поскольку Spring Boot и есть фреймворк Spring! Spring Boot можно рассматривать как новый способ простого и удобного создания приложений Spring.

Spring Boot упрощает способ разработки, поскольку позволяет легко создавать готовые к промышленной эксплуатации приложения на основе Spring, которые можно просто «запустить». Вы скоро узнаете, что с помощью Spring Boot можно создавать автономные приложения со встроенным сервером (по умолчанию Tomcat или Netty в случае использования новых *веб-реактивных*

модулей), полностью готовые к запуску и развертыванию. Мы поговорим больше про это в нескольких из последующих глав данной книги.

Одна из важнейших возможностей Spring Boot — продуманная среда выполнения, благодаря которой разработчику легче следовать рекомендуемым практикам по созданию надежных, расширяемых и масштабируемых приложений Spring.

Проект Spring Boot можно найти по адресу https://projects.spring.io/spring-boot/. А чрезвычайно подробную документацию — вот здесь: https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/. Домашняя страница Spring Boot показана на рис. 2.1.

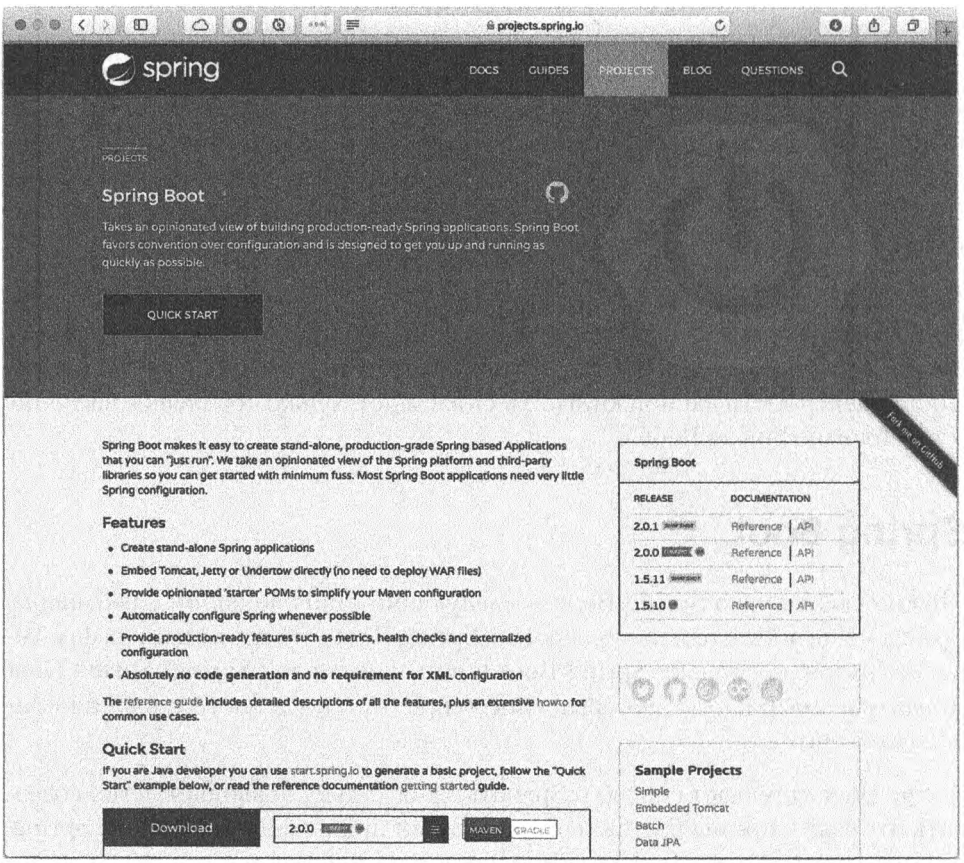


Рис. 2.1. Домашняя страница Spring Boot (https://projects.spring.io/spring-boot/)

Spring Boot спешит на помощь

Для создания приложений Spring необходимо знать все приемы настроек и/или требования данной технологии. Для запуска даже простейшего приложения Spring необходимо пройти длинный путь. Четыре года назад команда создателей Spring выпустила первую бета-версию Spring Boot, которую мне посчастливилось тестировать. Результаты были потрясающими. Теперь, после добавления в эту технологию множества новых возможностей, она и правда стала де-факто основным способом создания приложений Spring. Spring Boot существенно упрощает создание готовых к промышленной эксплуатации приложений.

На веб-странице проекта Spring Boot можно найти следующее утверждение: *Абсолютно никакой генерации кода и необходимости в XML-конфигурации.* Наверное, вы недоумеваете: как можно создавать приложения Spring и как они могут работать без какой-либо конфигурации? Контейнеру Spring нужно по крайней мере знать, как связать воедино классы, правда? А еще ему нужно знать, как использовать добавленные вами в приложение технологии. Что ж, не беспокойтесь. Я расскажу вам все секреты этой замечательной технологии. Но сначала создадим простейшее из возможных приложений Spring (листинг 2.1).

Листинг 2.1. app.groovy

```
@RestController
class WebApp{
      @GetMapping("/")
      String welcome(){
        "<h1><font face='verdana'>Spring Boot Rocks!</font></h1>"
     }
}
```

В листинге 2.1 приведено приложение Groovy — простейшее из возможных приложений Spring. Почему Groovy? Я всегда говорю студентам: если вы знаете Java, значит, знаете и Groovy. Groovy устраняет весь стереотипный код Java и позволяет создавать веб-приложения с помощью всего нескольких строк кода (но не волнуйтесь, эта книга в основном касается Java, лишь в последней главе мы поговорим о Groovy и Kotlin, новое прибавление к списку поддерживаемых Spring языков). Как же его запустить? Очень просто, всего лишь выполнить команду:

```
$ spring run app.groovy
```

После этого вы увидите вывод в консоль журнала с баннером Spring Boot, инициализацией контейнера Tomcat и примечание, что приложение было запущено на порте 8080. Если после этого открыть браузер и ввести `http://localhost:8080`, вы увидите текст Spring Boot Rocks!.

Наверное, вы воскликнете: «Погодите! Что за команда `spring run`? Как ее установить? Что еще мне нужно? Это и есть Spring Boot?» Ну, она представляет собой один из многих способов создания и запуска приложения Spring. Это была одна из первых попыток продемонстрировать мощь данной технологии (четыре года назад), простой сценарий, способный запустить полноценное веб-приложение Spring. Команда Spring Boot создала *интерфейс командной строки Spring Boot* (Spring Boot CLI).

Spring Boot CLI

Spring Boot CLI (интерфейс командной строки) — один из многих способов создания приложений Spring, впрочем обычно используемый для прототипов приложений. Можете считать его «песочницей» Spring Boot. Эталонная его модель описана в следующих разделах. Я просто хотел, чтобы вы попробовали на вкус немножко возможностей Spring с помощью простых скриптов Groovy или Java. Лично для меня интерфейс командной строки Spring — неотъемлемая составная часть экосистемы Spring Boot.

Вернемся к предыдущему коду. Обратили ли вы внимание, что в листинге 2.1 нет операторов импорта? Откуда же Spring Boot знает про веб-приложение вообще и как его запустить в частности?

Spring Boot CLI изучает код и пытается на основе аннотаций Spring MVC (`@RestController` и `@GetMapping`) выполнить его как веб-приложение, используя встроенный сервер Tomcat и запуская это веб-приложение на нем. Скрытая за кулисами магия состоит в том, что язык программирования Groovy обеспечивает простой способ перехвата операторов и динамического создания кода с помощью абстрактного синтаксического дерева (abstract syntax tree, AST); благодаря этому можно без проблем внедрить недостающий код Spring и выполнить его. Другими словами, Spring Boot CLI выясняет всю нужную информацию о вашем приложении и внедряет недостающие для создания и выполнения полноценного веб-приложения Spring части.

Помните, я упоминал, что он может также выполнять сценарии Java? Взглянем на Java-версию того же приложения. Пока что я просто покажу вам код;

если хотите выполнять эти приложения, загляните в приложение, где объясняется, как установить Spring Boot CLI, и описываются его возможности (листинг 2.2).

Листинг 2.2. SimpleWebApp.java

```
package com.apress.spring;

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;

@RestController
@SpringBootApplication
public class SimpleWebApp {

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

    @RequestMapping("/")
    public String greetings(){
        return "<h1>Spring Boot Rocks in Java too!</h1>";
    }
}
```

В листинге 2.2 показана точка входа для приложения Spring Boot на Java. Прежде всего в ней используются аннотация `@SpringBootApplication` и класс-одиночка (singleton) `SpringApplication` в методе `main`, который и выполняет приложение. Метод `SpringApplication.run` принимает два параметра. Первый параметр — основной класс конфигурации, содержащий аннотацию `@Configuration` (причем имя класса совпадает; но об этом позже). Второй параметр представляет собой аргументы приложения (о которых мы поговорим в следующих главах). Как вы видите из этой Java-версии, мы используем аннотации Spring MVC: `@RestController` и `@GetMapping`.

Запустить этот пример на выполнение можно с помощью команды:

```
$ spring run SimpleWebApp.java
```

Если после этого вы откроете браузер и перейдете по адресу http://localhost:8080, то увидите текст Spring Boot Rocks in Java too!.

Если вы хотите установить свой Spring Boot CLI, можете перейти к приложению данной книги, в котором пошагово описываются установка, все

возможности и преимущества Spring Boot CLI. Spring Boot CLI идеально подходит для быстрого создания прототипов облачных приложений Spring; именно поэтому я решил включить рассказ о нем в эту книгу.

Модель приложения Spring Boot

Spring Boot задает способ простого создания приложений Spring и модель программирования, следующую рекомендуемым практикам приложений Spring. Для создания приложения Spring необходимы следующие компоненты.

- Утилита сборки/управления зависимостями, например Maven или Gradle (Spring Boot также поддерживает *Ant* и *Ivy*; в этой книге для всех примеров достаточно Maven или Gradle).
- Корректное управление зависимостями и соответствующие плагины в утилите сборки. При работе с Maven требуется тег `<parent/>` (конечно, существуют и другие способы настройки Spring Boot, но использование тега `<parent/>` — простейший) и плагин `spring-boot-maven-plugin`. Если же вы используете Gradle, вам понадобятся плагины `org.springframework.boot` и `io.spring.dependency-management`.

Нужно также добавить необходимые зависимости с помощью `spring-boot-starters` и создать основной класс приложения, включающий:

- аннотацию `@SpringBootApplication`;
- оператор `SpringApplication.run` в методе `main`.

В следующем разделе мы собираемся создать наше первое приложение Spring Boot, и я дам подробные объяснения насчет всех вышеупомянутых компонентов. Все довольно просто, но с чего же начать? Существует ли утилита, с помощью которой можно начать проект Spring Boot? Ответ: да! Можно воспользоваться Spring Boot CLI, с помощью которого можно создавать проекты Spring Boot. Существуют также IDE (интерактивные среды разработки), например STS (Spring Tool Suite — «комплекс инструментов для Spring», https://spring.io/tools), IntelliJ IDEA от JetBrains (https://www.jetbrains.com/idea/), NetBeans (https://netbeans.org), Atom от GitHub (https://atom.io) и VSCode компании Microsoft (https://code.visualstudio.com). У Atom и VSCode есть плагины для упрощения разработки приложений Spring, но лично я, начиная проект Spring Boot, предпочитаю *Spring Initializr* (http://start.spring.io). В этой книге задействуется IntelliJ IDEA.

Взглянем, как использовать веб-сервис Spring Boot Initializr на примере создания нашего первого приложения Spring Boot.

Наше первое приложение Spring Boot. Для создания нашего первого приложения Spring Boot откройте браузер и перейдите на страницу http://start.spring.io (рис. 2.2).

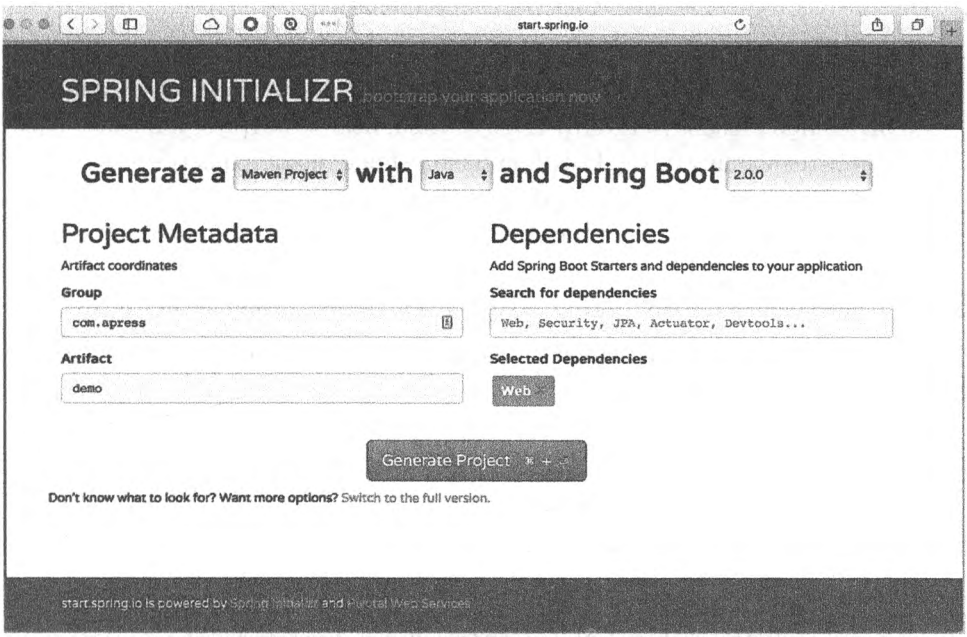


Рис. 2.2. http://start.spring.io

Рисунок 2.2 демонстрирует домашнюю страницу[1] Spring Boot Initializr, предоставляемого Pivotal веб-сервиса, с помощью которого можно легко создавать проекты Spring Boot.

1. Начнем с заполнения полей:
   - Group (Группа): `com.apress`;
   - Artifact (Артефакт): `demo`;
   - Dependencies (Зависимости): `web`.

[1] В настоящее время внешний вид этой страницы довольно сильно изменился, в том числе появились новые опции.

Можно выбрать тип проекта Maven или Gradle, язык программирования (Java, Groovy или Kotlin) и версию Spring Boot. Ниже расположена кнопка Generate Project (Сгенерировать проект), ниже которой есть ссылка Switch to the full version (Переключиться на полную версию). В данном случае можете ввести `Web` в поле Search for dependencies (Найти зависимости) и нажать Enter (см. рис. 2.2).

2. Нажмите кнопку Generate Project (Сгенерировать проект) и сохраните файл `demo.zip`.
3. Разархивируйте файл `demo.zip` и импортируйте проект в свою IDE (я предпочитаю использовать IntelliJ IDEA). Если вы посмотрите внимательнее, то увидите, что внутри ZIP-файла находится специальная обертка, какая именно — зависит от выбранного типа проекта. Если вы выбрали проект Gradle, это будет *gradlew* (обертка Gradle), если Maven — *mvnw* (обертка Maven). Это значит, что не нужно устанавливать никаких утилит сборки/управления, поскольку Spring Boot Initializr предоставляет все нужное.
4. Загляните в файлы управления сборкой/зависимостями. Откройте файл `pom.xml` или `build.gradle`.

   Если вы выбрали Maven, смотрите листинг 2.3.

Листинг 2.3. Файл pom.xml Maven

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                           http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>

      <groupId>com.apress</groupId>
      <artifactId>demo</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <packaging>jar</packaging>

      <name>demo</name>
      <description>Demo project for Spring Boot</description>

      <parent>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-parent</artifactId>
         <version>2.0.0.RELEASE</version>
         <relativePath/> <!-- lookup parent from repository -->
      </parent>
```

```
    <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
       <project.reporting.outputEncoding>UTF-8</project.reporting.
           outputEncoding>
      <java.version>1.8</java.version>
    </properties>

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

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

    <build>
      <plugins>
          <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
          </plugin>
      </plugins>
    </build>
</project>
```

Как видите, у нас уже есть часть основных компонентов: тег `<parent/>`, зависимость `spring-boot-starter-web` и плагин `spring-boot-maven-plugin`.

Если вы выбрали Gradle, смотрите листинг 2.4.

Листинг 2.4. build.gradle

```
buildscript {
    ext {
        springBootVersion = '2.0.0.RELEASE'
    }
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-
            plugin:${springBootVersion}")
    }
}
```

```
apply plugin: 'java'
apply plugin: 'eclipse'
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

group = 'com.apress'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = 1.8

repositories {
      mavenCentral()
}

dependencies {
   compile('org.springframework.boot:spring-boot-starter-web')
   testCompile('org.springframework.boot:spring-boot-starter-test')
}
```

В файле `build.gradle` тоже есть часть необходимых компонентов: плагины `org.springframework.boot` и `io.spring.dependency-management`, а также зависимость `spring-boot-starter-web`.

5. Откройте класс `com.apress.demo.DemoApplication.java` (листинг 2.5).

Листинг 2.5. Класс com.apress.demo.DemoApplication.java

```
package com.apress.demo;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

Здесь можно видеть остальные компоненты для запуска нашего приложения: аннотацию `@SpringBootApplication` и оператор `SpringApplication.run`.

6. Добавьте новый класс для веб-контроллера, отображающего текст. Создайте класс `com.apress.demo.WebController.java` (листинг 2.6).

Листинг 2.6. Класс com.apress.demo.WebController.java

```
package com.apress.demo;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
public class WebController {

    @GetMapping
    public String index(){
        return "Hello Spring Boot";
    }
}
```

Очень похоже на скрипт — просто `@RestController`, возвращающий строковое значение.

7. Для запуска приложения можно использовать IDE или перейти в корневой каталог приложения и выполнить следующую команду. Для Maven: `./mvnw spring-boot:run`, для Gradle: `./gradlew bootRun`.

   Далее можно зайти в браузер и перейти по адресу http://localhost:8080. При этом вы должны увидеть текст Hello Spring Boot.

Поздравляю! Вы только что создали свое первое приложение Spring Boot.

ПРИМЕЧАНИЕ

Весь прилагаемый к данной книге код можно найти на сайте издательства Apress. В нашем случае я создал два проекта: один для Maven и второй — для Gradle.

Почему Spring Boot?

Почему имеет смысл использовать Spring Boot? Это потрясающая технология, подходящая:

- для создания облачных нативных приложений, следующих паттернам 12 факторов, разработанным группой инженеров компании Netflix (https://12factor.net/ru/);
- повышения производительности труда разработчиков за счет уменьшения времени разработки и развертывания;
- создания готовых к промышленной эксплуатации корпоративных приложений Spring;

- нефункциональных требований, например Spring Boot Actuator (модуль, предоставляющий метрики для нового, не зависящего от платформы фреймворка *Micrometer* (https://micrometer.io), проверок состояния приложения и управления) и встраиваемых контейнеров для запуска веб-приложений (Tomcat, Netty, Undertow, Jetty и т. д.);
- микросервисов, которые все чаще привлекают внимание как основа для создания масштабируемых, высокодоступных и отказоустойчивых приложений. Spring Boot позволяет разработчикам сосредоточить свое внимание на одной бизнес-логике, оставляя наиболее трудную часть работы фреймворку Spring.

Возможности Spring Boot. Spring Boot обладает множеством возможностей. Мы поговорим о них подробнее в следующих главах, но часть из них мне хотелось бы описать в этом разделе.

- Spring Boot предоставляет класс `SpringApplication`. Я уже демонстрировал, что в Java-приложении Spring Boot метод `main` выполняет этот класс-одиночку. Данный класс предоставляет удобную возможность инициализации и запуска приложения Spring.
- Позволяет создавать приложения без какой-либо XML-конфигурации. Spring Boot не генерирует никакого кода.
- Предоставляет текучий API сборки посредством класса-одиночки `SpringApplicationBuilder`, с помощью которого можно создавать иерархии нескольких контекстов приложения. Эта конкретная возможность относится скорее к внутреннему устройству фреймворка Spring. Если вы разработчик на Spring, подождите до следующих глав, где я расскажу об этом подробнее, но если вы только начинаете знакомство со Spring и Spring Boot, то вам достаточно знать о возможности лучше контролировать свои приложения посредством расширения Spring Boot.
- Предоставляет дополнительные способы настройки событий и прослушивателей приложения Spring.
- Предоставляет продуманную технологию, нацеленную на создание «правильного» типа приложений, как веб-приложений (на основе встраиваемых контейнеров Tomcat, Netty, Undertow или Jetty), так и автономных.
- Включает интерфейс `org.springframework.boot.ApplicationArguments`, с помощью которого можно получить доступ к любым аргументам приложения.

Эта возможность может очень пригодиться, если требуется запускать приложение с параметрами.

- Позволяет выполнять код после запуска приложения. Нужно только реализовать интерфейс `CommandLineRunner` и предоставить реализацию метода `run(String ...args)`. Два конкретных примера использования этой возможности: инициализация записей в базе данных при запуске приложения и проверка перед выполнением приложения, работают ли сервисы.
- Позволяет использовать внешние конфигурации с помощью файлов `application.properties` и `application.yml`. Более подробно об этом — в следующих главах.
- Позволяет добавлять функциональные возможности, связанные с администрированием, обычно посредством JMX, с помощью свойства `spring.application.admin.enabled` в файле `application.properties` или `application.yml`.
- Предоставляет возможность применения *профилей*, благодаря которым приложение может работать в разных средах.
- Сильно упрощает настройку и использование журналирования.
- Предоставляет простой способ настройки зависимостей и управления ими с помощью стартовых POM. Другими словами, при необходимости создать веб-приложение достаточно только активировать зависимость `spring-boot-start-web` в файле `pom.xml` или `build.gradle`.
- Предоставляет готовые нефункциональные требования посредством использования Spring Boot Actuator с новым, не зависящим от платформы фреймворком *Micrometer*, позволяющим оснащать приложения средствами оценки производительности.
- Предоставляет аннотации `@Enable<возможность>`, упрощающие включение, настройку и использование таких технологий, как базы данных (SQL и NoSQL), кэширование, планирование выполнения задач и обмен сообщениями, Spring Integration, Spring Batch, Spring Cloud и многих других.

Spring Boot обладает этими и многими другими возможностями. Я расскажу о них подробнее в следующих главах. А пока пора узнать больше о внутреннем устройстве Spring Boot.

Резюме

В этой главе мы совершили краткий обзор технологии Spring Boot, предназначенной для простого создания готовых к промышленной корпоративной эксплуатации приложений Spring.

В следующих главах я покажу вам внутреннее устройство Spring Boot и скрытую «под капотом» магию по созданию «правильных» приложений на основе ваших зависимостей и кода. Мы поговорим обо всех потрясающих возможностях Spring Boot по мере создания различных проектов.

3 Внутреннее устройство и возможности Spring Boot

В предыдущей главе был приведен краткий обзор Spring Boot, основных компонентов, необходимых для создания приложения Spring Boot, и обсуждалось, насколько легко можно создавать проекты Spring Boot с помощью Spring Initializr.

В этой главе я расскажу вам, что происходит «за кулисами», когда Spring Boot запускает приложение. Главное здесь — автоконфигурация! Я начну со скриптов Groovy (опять же вы можете заглянуть в приложение и установить Spring Boot CLI). Мы будем работать с обычным проектом Java, как и в приложении Spring Boot из главы 2. Сначала разберемся, как работает автоматическая конфигурация.

Автоматическая конфигурация

Автоматическая конфигурация — одна из важнейших возможностей Spring Boot, поскольку именно она отвечает за настройку приложения Spring Boot в соответствии с путем к классам, аннотациями и всеми прочими объявлениями настроек, например с классами JavaConfig или XML.

В листинге 3.1 приведен тот же пример, что и в предыдущих главах, но теперь я воспользуюсь им для объяснения происходящего «за кулисами» при его запуске Spring Boot на выполнение.

Листинг 3.1. app.groovy

```
@RestController
class WebApp{

    @GetMapping('/')
    String index(){
        "Spring Boot Rocks"
   }
}
```

Запустите его с помощью Spring Boot CLI (интерфейса командной строки) командой:

```
$ spring run app.groovy
```

Spring Boot не генерирует никакого исходного кода, а добавляет его во время работы. Это одно из преимуществ Groovy: наличие доступа к AST (абстрактному синтаксическому дереву) во время выполнения. Spring Boot начинает с того, что импортирует, помимо прочих операций импорта, недостающие зависимости, например аннотацию `org.springframework.web.bind.annotation.RestController`.

Далее он определяет, что необходим *spring-boot-starter-web* (мы поговорим о нем подробнее в следующих главах), поскольку класс и метод снабжены аннотациями `@RestController` и `@GetMapping` соответственно. И добавляет в код аннотацию `@Grab("spring-boot-web-starter")` (удобно для импортов в сценариях Groovy).

Теперь Spring Boot добавляет аннотацию, запускающую автоконфигурацию, а именно — аннотацию `@EnableAutoConfiguration` (чуть позже мы поговорим об этой аннотации, в которой и заключается магия Spring Boot), а затем добавляет метод `main` — точку входа приложения. Получившийся в итоге код приведен в листинге 3.2.

Листинг 3.2. Файл app.groovy, модифицированный Spring Boot

```
import org.springframework.web.bind.annotation.RestController
// Прочие операции импорта

@Grab("spring-boot-web-starter")
@EnableAutoConfiguration
@RestController
class WebApp{
   @GetMapping("/")
    String greetings(){
        "Spring Boot Rocks"
   }
```

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

В листинге 3.2 показана модифицированная программа, которую фактически выполняет Spring Boot. Запустив листинг 3.1 с параметром `--debug`, можно увидеть в действии работу автоконфигурации:

```
$ spring run app.groovy --debug
...
DEBUG 49009 --- [] autoConfigurationReportLoggingInitializer :
=========================
AUTO-CONFIGURATION REPORT
=========================
Positive matches:
-----------------
// Здесь будут указаны все условия, которые были удовлетворены
// при обеспечении запуска веб-приложения. А все благодаря
// аннотации @RestController.

Negative matches:
-----------------
// Все условия, которые не были удовлетворены. Например, вы увидите,
// что условие для класса ActiveMQAutoConfiguration не удовлетворяется
// из-за отсутствия какой-либо ссылки на ActiveMQConnectionFactory.
```

Посмотрите на выведенное в терминал в результате выполнения этой команды. Обратите внимание на все успешные (`Positive matches`) и неудачные (`Negative matches`) операции проверки условий, произведенные Spring Boot при запуске этого простого приложения. При использовании командной строки Spring Boot последний выполняет массу работы, пытаясь угадать, какой именно вид приложения мы хотим запустить. А когда мы создаем проект Maven или Gradle и указываем зависимости (в файлах `pom.xml` или `build.gradle` соответственно), то помогаем Spring Boot определить это на основе зависимостей.

Отключение конкретных автоконфигурационных классов

В главе 2 я упоминал аннотацию `@SpringBootApplication` — один из главных компонентов приложения Spring Boot. Эта аннотация эквивалентна объявлению аннотаций `@Configuration`, `@ComponentScan` и `@EnableAutoConfiguration`. Почему я упоминаю этот факт? А потому, что существует возможность отключения конкретного автоконфигурационного класса путем добавления параметра

exclude при использовании в своем классе аннотаций @EnableAutoConfiguration или @SpringBootApplication. Взглянем в листинге 3.3 на пример в сценарии Groovy.

Листинг 3.3. Файл app.groovy

```
import org.springframework.boot.autoconfigure.jms.activemq.
ActiveMQAutoConfiguration

@RestController
@EnableAutoConfiguration(exclude=[ActiveMQAutoConfiguration.class])
class WebApp{

    @RequestMapping("/")
        String greetings(){
            "Spring Boot Rocks"
        }
}
```

Листинг 3.3 демонстрирует аннотацию @EnableAutoConfiguration с параметром exclude. В качестве значения этого параметра указывается массив автоконфигурационных классов. Если запустить этот пример с помощью следующей команды, вы увидите соответствующую запись в разделе Exclusions (Исключенные):

```
$ spring run app.groovy --debug
...
Exclusions:
-----------
    org.springframework.boot.autoconfigure.jms.activemq.
        ActiveMQAutoConfiguration
...
```

Эта методика очень удобна для скриптов Groovy, если необходимо, чтобы Spring Boot пропустил конкретные ненужные автоконфигурации.

Посмотрим, как использовать это для Java-приложения Spring Boot (листинг 3.4).

Листинг 3.4. DemoApplication.java: фрагмент кода Spring Boot

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration;
```

```
import org.springframework.boot.autoconfigure.jms.activemq.
    ActiveMQAutoConfiguration;

@SpringBootApplication(exclude={ActiveMQAutoConfiguration.class,
    DataSourceAutoConfiguration.class})
public class DemoApplication {

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

В листинге 3.4 показана Java-версия; в этом примере основной класс объявляется только с аннотацией `@SpringBootApplication`, внутри которой можно исключить любые ненужные автоконфигурационные классы. В листинге 3.4 показано исключение двух классов: `ActiveMQAutoConfiguration` и `DataSourceAutoConfiguration`. Почему мы не используем аннотацию `@EnableAutoConfiguration`? Как вы помните, дело в том, что аннотация `@SpringBootApplication` наследует `@EnableAutoConfiguration`, `@Configuration` и `@ComponentScan`, поэтому можно применить параметр `exclude` в аннотации `@SpringBootApplication`.

При запуске проекта Maven или Gradle (с примером из листинга 3.4) с опцией отладки в консоль будет выведено примерно следующее:

```
...
Exclusions:
-----------
   org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration
   org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration
...
```

Аннотации @EnableAutoConfiguration и @Enable<технология>

Фреймворк Spring и некоторые из его модулей, например Spring Data, Spring AMQP и Spring Integration, предоставляют аннотации `@Enable<технология>`; например, частью упомянутых модулей являются аннотации `@EnableTransactionManagement`, `@EnableRabbit` и `@EnableIntegration`. С помощью этих аннотаций в приложениях Spring можно следовать паттерну *«соглашения важнее конфигурации»* (convention over configuration), упрощающему разработку и сопровождение приложений, без необходимости волноваться слишком сильно об их конфигурации.

Spring Boot пользуется преимуществами этих аннотаций, применяемых внутри аннотации `@EnableAutoConfiguration` для выполнения автоконфигурации. Взглянем повнимательнее на аннотацию `@EnableAutoConfiguration`, разберемся в логике ее работы и поймем, как в эту схему вписываются аннотации `@Enable<технология>` (листинг 3.5).

Листинг 3.5. org.springframework.boot.autoconfigure.EnableAutoConfiguration.java

```
...
@AutoConfigurationPackage
@Import(AutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {

      Class<?>[] exclude() default {};

      String[] excludeName() default {};

}
```

В листинге 3.5 показана аннотация `@EnableAutoConfiguration`; как вы уже знаете, этот класс пытается произвести настройки компонентов, которые, вероятно, понадобятся вашему приложению. Применение автоконфигурационных классов происходит в соответствии с *путем к классам* (classpath) и описанными в приложении компонентами, но еще более расширяет возможности класс `org.springframework.boot.autoconfigure.AutoConfigurationImportSelector`, предназначенный для поиска всех необходимых классов конфигурации.

У класса `AutoConfigurationImportSelector` есть несколько методов, самый важный из которых для автоконфигурации — метод `getCandidateConfigurations` (листинг 3.6).

Листинг 3.6. Фрагмент кода класса
org.springframework.boot.autoconfigure.AutoConfigurationImportSelector

```
...
protected List<String> getCandidateConfigurations(
                              AnnotationMetadata metadata,
                              AnnotationAttributes attributes) {
  List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
            getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
      Assert.notEmpty(configurations,
      "No auto configuration classes found in META-INF/spring.factories.
      If you are using a custom packaging, make sure that file is correct.");
```

```
        return configurations;
}
```

В листинге 3.6 приведен фрагмент кода класса `org.springframework.boot.autoconfigure.AutoConfigurationImportSelector`, а именно метод `getCandidateConfigurations`, возвращающий результат вызова `SpringFactoriesLoader.loadFactoryNames`. Метод `SpringFactoriesLoader.loadFactoryNames` производит поиск файла `META-INF/spring.factories`, описанного в JAR-файле `spring-boot-autoconfigure` (его содержимое приведено в листинге 3.7).

Листинг 3.7. Фрагмент файла spring-boot-autoconfigure-<version>.jar/META-INF/spring.factories

```
# Инициализаторы
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.
SharedMetadataReaderFactoryContextInitializer,\
org.springframework.boot.autoconfigure.logging.
ConditionEvaluationReportLoggingListener

# Прослушиватели приложения
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer

# Классы автоконфигурации
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.
    SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.
    PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.
    CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration,\
....
....
```

Как видно из листинга 3.7, в файле `spring.factories` описываются все автоконфигурационные классы, используемые для задания всех необходимых для работы приложения параметров конфигурации. Взглянем на класс `CloudAutoConfiguration` (листинг 3.8).

Листинг 3.8. org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration.java

```
package org.springframework.boot.autoconfigure.cloud;

import org.springframework.boot.autoconfigure.AutoConfigureOrder;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure.condition.
    ConditionalOnMissingBean;
import org.springframework.boot.autoconfigure.condition.
    ConditionalOnProperty;
import org.springframework.cloud.Cloud;
import org.springframework.cloud.app.ApplicationInstanceInfo;
import org.springframework.cloud.config.java.CloudScan;
import org.springframework.cloud.config.java.CloudScanConfiguration;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Import;
import org.springframework.context.annotation.Profile;
import org.springframework.core.Ordered;

@Configuration
@Profile("cloud")
@AutoConfigureOrder(CloudAutoConfiguration.ORDER)
@ConditionalOnClass(CloudScanConfiguration.class)
@ConditionalOnMissingBean(Cloud.class)
@ConditionalOnProperty(prefix = "spring.cloud", name = "enabled",
                       havingValue = "true", matchIfMissing = true)
@Import(CloudScanConfiguration.class)
public class CloudAutoConfiguration {

      // Облачная конфигурация должна указываться как можно раньше
      // (перед данными, mongo и т. д.)
      public static final int ORDER = Ordered.HIGHEST_PRECEDENCE + 20;
}
```

В листинге 3.8 показан класс `CloudAutoConfiguration`. Как вы можете видеть, это очень короткий класс, который тем не менее производит настройку облачного приложения, если находит по пути к классам приложения классы `spring.cloud`. Но как он это делает? Он задействует аннотации `@ConditionalOnClass` и `@ConditionalOnMissingBean` для того, чтобы определить, является приложение облачным или нет. Не задумывайтесь пока об этом, вам еще предстоит использовать эти аннотации при создании вашего собственного автоконфигурационного класса в главе 13 «Расширение возможностей Spring Boot».

Еще в листинге 3.8 следует отметить использование аннотации `@ConditionalOnProperty`, применяемой, только если активировано свойство `spring.cloud`.

Стоит отметить, что данная автоконфигурация производится в профиле `cloud`, указанном в аннотации `@Profile`. Аннотация `@Import` применяется только в том случае, если удовлетворяются условия остальных аннотаций (поскольку здесь используются аннотации `@Conditional*`), то есть импорт класса `CloudScanConfiguration` выполняется, если классы `spring-cloud-*` находятся по пути к классам. Более подробно этот вопрос описан в главе 13. Пока необходимо только понимать, что автоконфигурация использует путь к классам, чтобы определить, какие настройки необходимы для приложения. Именно поэтому мы называем Spring Boot *продуманной средой выполнения*, помните?

Возможности Spring Boot

В этом разделе я покажу вам некоторые возможности Spring Boot. Spring Boot можно чрезвычайно широко настраивать под свои потребности, начиная от автоконфигурации для настройки приложения (на основе пути к классам) и до выбора способа его запуска, отображаемых, активируемых и отключаемых элементов на основе его собственных свойств. Так что изучим свойства Spring Boot, с помощью которых можно настроить под свои нужды свое приложение Spring.

Создадим Java-проект Spring Boot с помощью Spring Boot Initializr. Откройте браузер и перейдите по адресу https://start.spring.io. Внесите в поля указанные ниже значения. Не забудьте щелкнуть на поле Switch to the full version (Переключиться на полную версию)[1], чтобы указать название пакета.

- Group (Группа): `com.apress.spring`.
- Artifact (Артефакт): `spring-boot-simple`.
- Name (Название): `spring-boot-simple`.
- Package Name (Название пакета): `com.apress.spring`.

Можете выбрать тип проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте полученный ZIP-файл.

[1] Как уже упоминалось в примечании выше, интерфейс Spring Boot Initializr несколько изменился, роль Switch to the full version теперь играет раскрывающееся меню Options (Опции).

Разархивируйте его туда, куда вам удобно, и импортируйте его в удобный вам IDE (рис. 3.1).

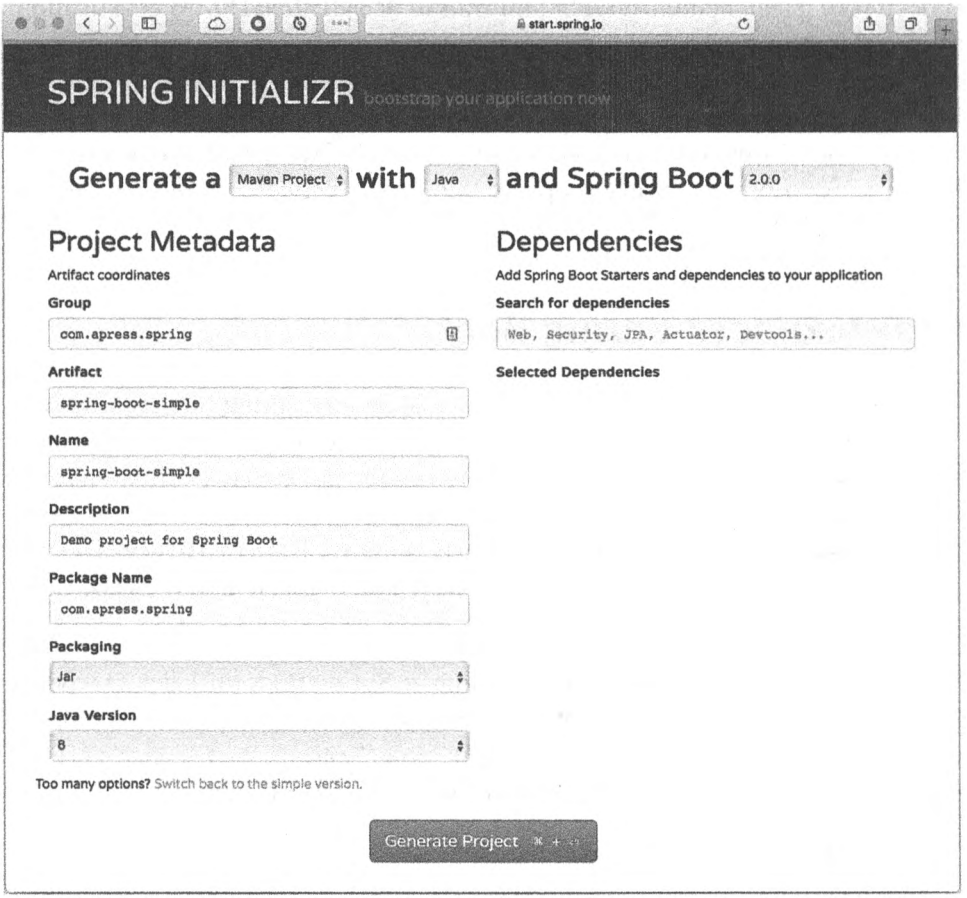


Рис. 3.1. Проект Spring Boot

ПРИМЕЧАНИЕ

Вы можете скачать исходный код для данной книги с сайта Apress, и в каждом из проектов вы найдете файл pom.xml для Maven и build.gradle для Gradle, так что вы можете выбрать ту утилиту сборки, какая вам больше нравится.

Теперь запустите приложение Spring Boot. Воспользуйтесь IDE или откройте терминал и выполните следующую команду, если вы применяете Maven.

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

Если же вы используете Gradle, можете выполнить:

```
$ ./gradlew bootRun
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.0.RELEASE)
INFO 10669 --- [    main] c.a.spring.SpringBootSimpleApplication   :
Starting SpringBootSimpleApplication on ...
INFO 10669 --- [    main] c.a.spring.SpringBootSimpleApplication   : No
active profile set, falling back to default profiles: default
INFO 10669 --- [    main] s.c.a.AnnotationConfigApplicationContext :
Refreshing org.springframework.context.annotation...
INFO 10669 --- [    main] o.s.j.e.a.AnnotationMBeanExporter        :
Registering beans for JMX exposure on startup
INFO 10669 --- [    main] c.a.spring.SpringBootSimpleApplication   : Started
SpringBootSimpleApplication in 1.582 seconds (JVM running for 4.518)
INFO 10669 --- [Thread-3] s.c.a.AnnotationConfigApplicationContext :
Closing org.springframework.context.annotation...
INFO 10669 --- [Thread-3] o.s.j.e.a.AnnotationMBeanExporter        :
Unregistering JMX-exposed beans on shutdown
```

При этом вы должны увидеть нечто похожее на данный вывод. В нем можно видеть баннер (Spring Boot) и журнал. Взглянем на основное приложение в листинге 3.9.

Листинг 3.9. src/main/java/com/apress/spring/SpringBootSimpleApplication.java

```
package com.apress.spring;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class SpringBootSimpleApplication {

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

В листинге 3.9 приведено основное приложение. Компоненты Spring Boot уже известны вам из предыдущей главы, но опишем их еще раз:

- `@SpringBootApplication`. Фактически эта аннотация представляет собой аннотации `@ComponentScan`, `@Configuration` и `@EnableAutoConfiguration`. Про аннотацию `@EnableAutoConfiguration` вам уже все известно из предыдущих разделов;
- `SpringApplication`. Этот класс предоставляет первоначальный загрузчик приложения Spring Boot, выполняемый в методе `main`. Необходимо только передать выполняемый класс.

Теперь мы готовы начать настройку приложения Spring Boot под свои нужды.

Класс SpringApplication

Класс `SpringApplication` дает возможность более продвинутой настройки путем создания его экземпляра и выполнения намного более расширенных действий (листинг 3.10).

Листинг 3.10. src/main/java/com/apress/spring/SpringBootSimpleApplication.java, версия 2

```
package com.apress.spring;

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

@SpringBootApplication
public class SpringBootSimpleApplication {

    public static void main(String[] args) {
      SpringApplication app =
            new SpringApplication(SpringBootSimpleApplication.class);
            // Здесь можно добавить дополнительные возможности
            app.run(args);
      }
}
```

Класс `SpringApplication` позволяет настраивать поведение приложения и контролировать основной объект `ApplicationContext`, в котором используются все компоненты. Если вы хотите узнать больше об интерфейсе `ApplicationContext` и его использовании — рекомендую упоминавшуюся выше книгу «Spring 5 для профессионалов», в которой авторы объясняют все о Spring. В данном случае мы сосредоточим наше внимание на отдельных возможностях Spring Boot. Начнем с кое-чего по-настоящему интересного.

Пользовательский баннер

При каждом запуске приложения в начале журнала отображается баннер. Существует несколько способов его настройки.

Реализуем интерфейс `org.springframework.boot.Banner` (листинг 3.11).

Листинг 3.11. src/main/java/com/apress/spring/SpringBootSimpleApplication.java, версия 3

```
package com.apress.spring;

import java.io.PrintStream;

import org.springframework.boot.Banner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.core.env.Environment;

@SpringBootApplication
public class SpringBootSimpleApplication {

    public static void main(String[] args) {
       SpringApplication app = new SpringApplication
           (SpringBootSimpleApplication.class);
       app.setBanner(new Banner() {
            @Override
            public void printBanner(Environment environment,
                 Class<?> sourceClass, PrintStream out) {
            out.print("\n\n\tThis is my own banner!\n\n".toUpperCase());
            }
      });
               app.run(args);
    }
}
```

При запуске этой версии приложения вы увидите что-то вроде:

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

      THIS IS MY OWN BANNER!

INFO[main] c.a.spring.SpringBootSimpleApplication   : Starting
SpringBootSimpleApplication ...
...
...
INFO[main] c.a.spring.SpringBootSimpleApplication   : Started
SpringBootSimpleApplication in 0.789seconds (JVM running for 4.295)
```

```
INFO[Th-1] s.c.a.AnnotationConfigApplicationContext : Closing org.
springframework.context.annotation.AnnotationConfigApplicationContext@203f
6b5: startup date [Thu Feb 25 19:00:34 MST 2016]; root of context hierarchy
INFO[Th-1] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-
exposed beans on shutdown
```

Можете создать свой собственный ASCII-баннер и отобразить его. Как? Существует замечательный сайт, преобразующий текст в псевдографику (http://patorjk.com), как показано на рис. 3.2.

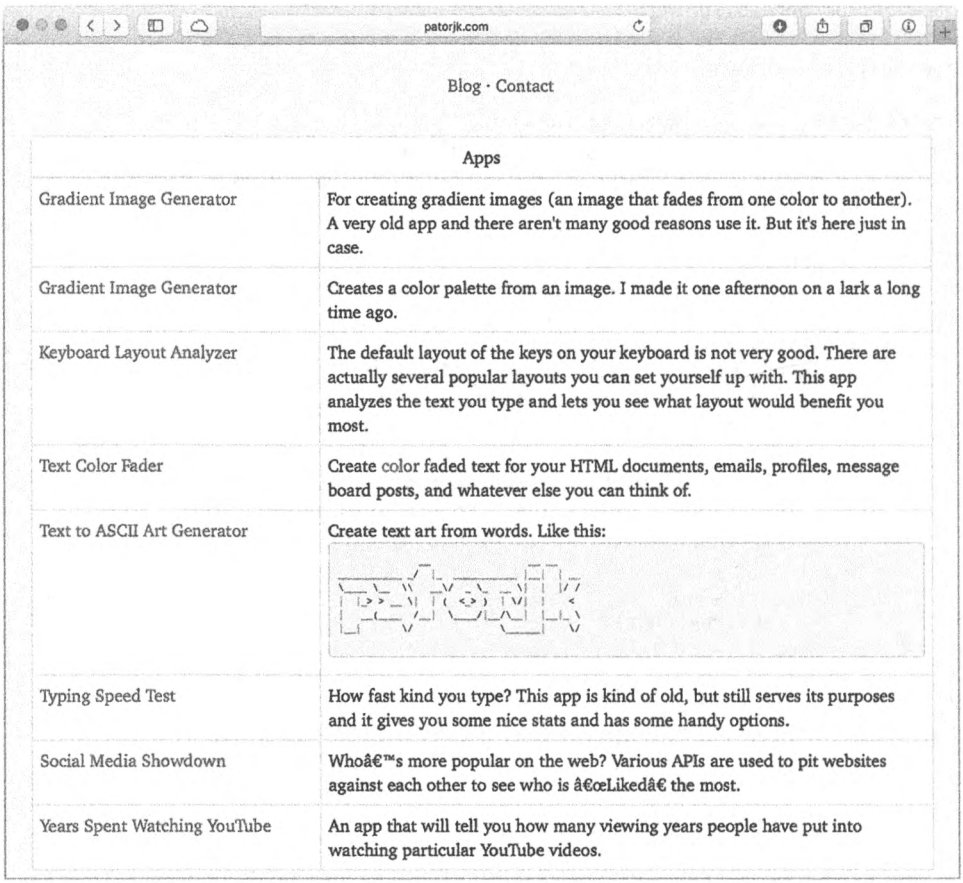


Рис. 3.2. Генератор псевдографики из текста

На рис. 3.2 показан веб-сайт http://patorjk.com. Щелкните на ссылке Text to ASCII Art Generator (Генератор псевдографики из текста). Далее введите в текстовое

поле `Pro Spring Boot 2.0` (или любой другой текст). Теперь щелкните на пункте Test All (Проверить все), чтобы увидеть все варианты псевдографики (рис. 3.3).

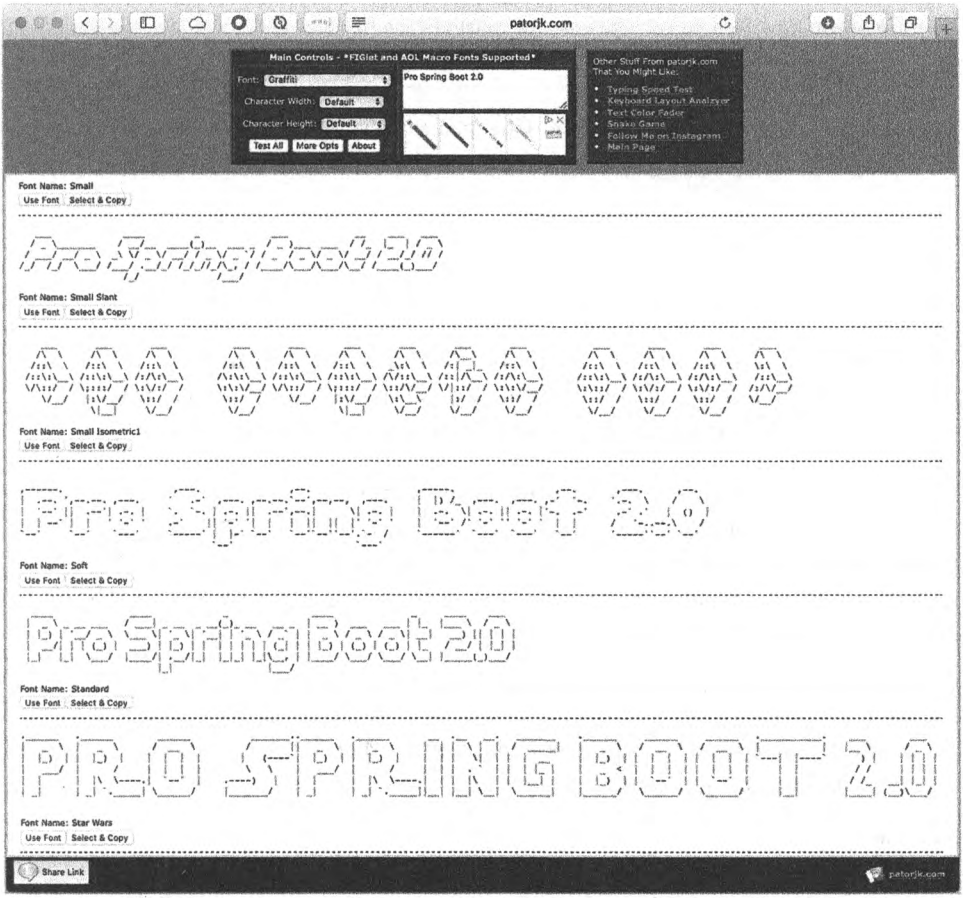


Рис. 3.3. Псевдографика

Рисунок 3.3 демонстрирует все варианты псевдографики (около 314). Теперь можно выбрать один из них. Нажмите кнопку Select Text (Выбрать текст), скопируйте текст (Ctrl+C в Windows/Cmd+C в macOS)[1] и создайте файл с именем `banner.txt` в каталоге `src/main/resources/` (рис. 3.4).

[1] Сейчас этот процесс немного упростился, возле каждого из вариантов псевдографики располагается кнопка Select & Copy (Выбрать и скопировать).

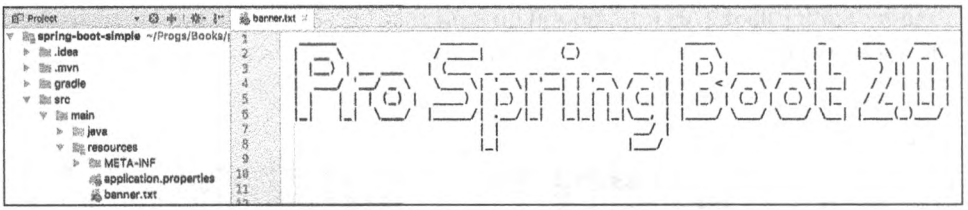


Рис. 3.4. Содержимое файла src/main/resources/banner.txt

Теперь можете запустить приложение снова:

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

Если запустить приложение с использованием листинга 3.11, то заданный в коде текстовый баннер будет перекрыт файлом `banner.txt`, расположенным по пути к классам (classpath); таково поведение по умолчанию.

По умолчанию Spring Boot ищет файл `banner.txt` по пути к классам. Но можно изменить его расположение. Создайте еще один файл `banner.txt` (или скопируйте уже созданный) в каталоге `src/main/resources/META-INF/`. Запустите приложение еще раз с параметром `-D`. Если вы используете Maven, выполняемая команда должна выглядеть следующим образом:

```
$ ./mvnw spring-boot:run -Dspring.banner.location=classpath:/META-INF/banner.txt
```

Если вы используете Gradle, то сначала нужно добавить эту конфигурацию в конец файла `build.gradle`.

```
bootRun {
    systemProperties = System.properties
}
```

Выполните следующую команду:

```
$ ./gradlew bootRun -Dspring.banner.location=classpath:/META-INF/banner.txt
```

В этой команде используется флаг `-D` для передачи свойства `spring.banner.location`, указывающего на новое местоположение пути к классам, `/META-INF/banner.txt`. Это свойство можно объявить в файле `src/main/resources/application.properties`, как показано ниже:

```
spring.banner.location=classpath:/META-INF/banner.txt
```

Если вы используете Maven, запустите его следующим образом:

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

Если вы используете Gradle, запустите его следующим образом:

```
$ ./gradlew bootRun
```

Существует несколько вариантов настройки файла `banner.txt`.

Можно полностью убрать баннер, объявив его в `src/main/resources/application.properties` вот так:

```
spring.main.banner-mode=off
```

У этой команды есть приоритет над файлом `banner.txt` по умолчанию, расположенным по пути `classpath:banner.txt`. Можно также сделать это программным образом (только не забудьте закомментировать свойство) (листинг 3.12).

Листинг 3.12. src/main/java/com/apress/spring/SpringBootSimpleApplication.java, версия 4

```
package com.apress.spring;

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

@SpringBootApplication
public class SpringBootSimpleApplication {

      public static void main(String[] args) {
            SpringApplication app = new SpringApplication
                (SpringBootSimpleApplication.class);
            app.setBannerMode(Banner.Mode.OFF);
            app.run(args);
      }
}
```

Класс SpringApplicationBuilder

Класс `SpringApplicationBuilder` предоставляет текучий API и представляет собой построитель экземпляров `SpringApplication` и `ApplicationContext`. Он также поддерживает иерархию; все продемонстрированное выше (с помощью `SpringApplication`) можно сделать с его помощью (листинг 3.13).

Листинг 3.13. src/main/java/com/apress/spring/SpringBootSimpleApplication.java, версия 5

```
package com.apress.spring;

import org.springframework.boot.Banner;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.builder.SpringApplicationBuilder;

@SpringBootApplication
public class SpringBootSimpleApplication {

      public static void main(String[] args) {
        new SpringApplicationBuilder()
            .bannerMode(Banner.Mode.OFF)
            .sources(SpringBootSimpleApplication.class)
            .run(args);
      }
}
```

Листинг 3.13 демонстрирует текучий API класса `SpringApplicationBuilder`. Рассмотрим еще несколько примеров.

При создании приложения Spring возможна иерархия контекстов приложения (если хотите узнать больше о контексте приложения в Spring, рекомендую книгу *Pro Spring 5* («Spring 5 для профессионалов»)). Создать иерархию можно с помощью класса `SpringApplicationBuilder`.

```
new SpringApplicationBuilder(SpringBootSimpleApplication.class)
            .child(OtherConfig.class)
            .run(args);
```

В случае веб-конфигурации ее следует объявлять как дочерний узел. Кроме того, у родительского и дочернего узлов должен быть один интерфейс `org.springframework.core.env.Environment` (представляющий среду, в которой работает текущее приложение; он связан с объявлениями профилей и свойств).

Можно также журналировать информацию о запуске приложения; по умолчанию она журналируется (установлено в `true`).

```
new SpringApplicationBuilder(SpringBootSimpleApplication.class)
            .logStartupInfo(false)
            .run(args);
```

Можно активировать профили.

```
new SpringApplicationBuilder(SpringBootSimpleApplication.class)
            .profiles("prod","cloud")
            .run(args);
```

Чуть позже я расскажу вам о профилях, и вы поймете, что означают предыдущие строки кода.

Можно подключать прослушиватели для некоторых из событий `ApplicationEvent`.

```
Logger log = LoggerFactory.getLogger(SpringBootSimpleApplication.class);
new SpringApplicationBuilder(SpringBootSimpleApplication.class)
    .listeners(new ApplicationListener<ApplicationEvent>() {
      @Override
      public void onApplicationEvent(ApplicationEvent event) {
          log.info("#### > " +  event.getClass().getCanonicalName());
      }
    })
    .run(args);
```

В этом случае при запуске приложения должно быть выведено следующее.

```
...
#### > org.springframework.boot.context.event.ApplicationPreparedEvent
...
#### > org.springframework.context.event.ContextRefreshedEvent
#### > org.springframework.boot.context.event.ApplicationReadyEvent
...
#### > org.springframework.context.event.ContextClosedEvent
...
```

В приложение можно также добавить нужную для обработки этих событий логику. Можно также использовать следующие дополнительные события: `ApplicationStartedEvent` (отправляется при запуске приложения), `ApplicationEnvironmentPreparedEvent` (отправляется, когда становится известной среда), `ApplicationPreparedEvent` (отправляется после определений компонентов), `ApplicationReadyEvent` (отправляется, когда приложение готово к работе), `ApplicationFailedEvent` (отправляется в случае возникновения исключений во время запуска приложения). Некоторые из них показаны в предыдущем выводе (относящиеся скорее к контейнеру Spring).

Существует возможность остановить выполнение любой автоконфигурации веб-среды. Как вы помните, Spring Boot «догадывается», какой тип приложения вы запускаете, основываясь на пути к классам. Для веб-приложения алгоритм очень прост; но представьте себе, что вы используете библиотеки, работающие без веб-среды, а ваше приложение — не веб-приложение, а Spring Boot задаст его конфигурацию как веб-приложения.

```
new SpringApplicationBuilder(SpringBootSimpleApplication.class)
            .web(WebApplicationType.NONE)
            .run(args);
```

Оказывается, что `WebApplicationType.NONE` — значение перечисляемого типа. Приложение может иметь тип `WebApplicationType.NONE`, `WebApplicationType.SERVLET` или `WebApplicationType.REACTIVE`. Как видите, смысл этих значений вполне очевиден.

Аргументы приложения

Spring Boot позволяет передавать приложению аргументы. `SpringApplication.run(SpringBootSimpleApplication.class, args)` дает возможность доступа к аргументам в компонентах (листинг 3.14).

Листинг 3.14. src/main/java/com/apress/spring/SpringBootSimpleApplication.java, версия 6

```
package com.apress.spring;

import java.io.IOException;
import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.stereotype.Component;

@SpringBootApplication
public class SpringBootSimpleApplication {
      public static void main(String[] args) throws IOException {
            SpringApplication.run(SpringBootSimpleApplication.class, args);
      }
}

@Component
class MyComponent {

    private static final Logger log = LoggerFactory.getLogger
        (MyComponent.class);

    @Autowired
    public MyComponent(ApplicationArguments args) {
        boolean enable = args.containsOption("enable");
        if(enable)
            log.info("## > You are enabled!");
```

```
        List<String> _args = args.getNonOptionArgs();
           log.info("## > extra args ...");
        if(!_args.isEmpty())
           _args.forEach(file -> log.info(file));
    }
}
```

При выполнении метода `containsOption` ожидается аргумент вида `--<аргумент>`. В листинге 3.14 он ожидает аргумент `--enable`, остальные аргументы принимает метод `getNonOptionArgs`. Чтобы попробовать их работу, можете выполнить следующую команду:

```
$ ./mvnw spring-boot:run -Dspring-boot.run.arguments="--enable"
```

В результате должна быть выведена надпись `## > You are enabled`.

Можно запустить подобный код и вот так:

```
$ ./mvnw spring-boot:run -Dspring-boot.run.arguments="arg1,arg2"
```

При использовании Gradle (по состоянию на момент написания данной книги) придется немного подождать, поскольку с передачей параметров все еще существуют определенные проблемы (см. https://github.com/spring-projects/spring-boot/issues/1176); впрочем, как описывается в следующем разделе, можно передавать параметры в исполняемый JAR-файл.

Передача параметров в исполняемый JAR

Один из вариантов: создать автономное приложение — исполняемый JAR-файл (позднее мы поговорим об этом подробнее). Для создания исполняемого JAR-файла выполните следующую команду (если используете Maven):

```
$ ./mvnw package
```

Эта команда создает исполняемый JAR-файл в каталоге `target`.

Или, если вы задействуете Gradle, воспользуйтесь следующей командой:

```
$./gradlew build
```

Эта команда создает исполняемый JAR-файл в каталоге `build/libs`.

Теперь можете запустить этот исполняемый JAR.

```
$ java -jar spring-boot-simple-0.0.1-SNAPSHOT.jar
```

Передавать аргументы можно следующим образом:

```
$ java -jar spring-boot-simple-0.0.1-SNAPSHOT.jar --enable arg1 arg2
```

При этом в консоль должен быть выведен вышеупомянутый текст для аргумента `enable` и список аргументов `arg1` и `arg2`.

Интерфейсы ApplicationRunner и CommandLineRunner

Если вы заметили, Spring Boot после выполнения приложения `SpringApplication` завершает работу. Использовать компоненты после завершения выполнения данного класса нельзя, но существует решение этой проблемы. В Spring Boot есть интерфейсы `ApplicationRunner` и `CommandLineRunner`, предоставляющие метод `run` (листинг 3.15).

Листинг 3.15. src/main/java/com/apress/spring/SpringBootSimpleApplication.java, версия 7

```
package com.apress.spring;

import java.io.IOException;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.ApplicationArguments;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class SpringBootSimpleApplication implements CommandLineRunner,
ApplicationRunner{
      private static final Logger log = LoggerFactory.getLogger
           (SpringBootSimpleApplication.class);

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

      @Bean
      String info(){
```

```
            return "Just a simple String bean";
      }

      @Autowired
      String info;

      @Override
      public void run(ApplicationArguments args) throws Exception {
            log.info("## > ApplicationRunner Implementation...");
            log.info("Accessing the Info bean: " + info);
            args.getNonOptionArgs().forEach(file -> log.info(file));
      }
      @Override
      public void run(String... args) throws Exception {
            log.info("## > CommandLineRunner Implementation...");
            log.info("Accessing the Info bean: " + info);
            for(String arg:args)
                  log.info(arg);
      }
}
```

В листинге 3.15 приведены интерфейсы `ApplicationRunner` и `CommandLineRunner` и их реализации. Интерфейс `CommandLineRunner` предоставляет метод `public void run(String... args)`, а `ApplicationRunner` — метод `public void run(ApplicationArguments args)`. По сути, это одно и то же. Реализовывать оба одновременно не нужно. Листинг 3.16 демонстрирует еще один способ использования интерфейса `CommandLineRunner`.

Листинг 3.16. src/main/java/com/apress/spring/SpringBootSimpleApplication.java, версия 8

```
package com.apress.spring;

import java.io.IOException;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class SpringBootSimpleApplication {
      private static final Logger log = LoggerFactory.getLogger
            (SpringBootSimpleApplication.class);
      public static void main(String[] args) throws IOException {
```

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

    @Bean
    String info(){
        return "Just a simple String bean";
    }

    @Autowired
    String info;

    @Bean
    CommandLineRunner myMethod(){
        return args -> {
            log.info("## > CommandLineRunner Implementation...");
            log.info("Accessing the Info bean: " + info);
            for(String arg:args)
                log.info(arg);
        };
    }
}
```

В листинге 3.16 показан снабженный аннотацией `@Bean` метод, возвращающий реализацию интерфейса `CommandLineRunner`. Для возврата в этом примере используется синтаксис Java 8 (лямбда-выражение). Можно добавить столько возвращающих `CommandLineRunner` методов, сколько вам нужно. Если вы хотите выполнять их в определенном порядке, можете воспользоваться аннотацией `@Order`.

Конфигурация приложения

Мы, разработчики, хорошо знаем, что избавиться от конфигурационных настроек приложений не удастся никогда. Мы всегда ищем, где бы сохранить примеры URL, IP-адреса, учетные данные, информацию о базе данных и т. д. — сведения, которые обычно очень часто используются в приложениях. Мы знаем, что не рекомендуется «зашивать» подобную информацию в приложение. Ее нужно вынести за его пределы для безопасности, а также для удобства применения и развертывания.

В случае Spring можно использовать либо XML и тег `<context:property-placeholder/>`, либо аннотацию `@PropertySource`, указывающую на файл, где объявлены свойства. Spring Boot предоставляет тот же механизм, но с дополнительными усовершенствованиями.

Spring Boot предоставляет различные варианты сохранения конфигурации приложения.

- Можно использовать файл `application.properties`, который должен располагаться в корневом каталоге пути к классам приложения (существуют и другие варианты расположения этого файла, которые я покажу вам позже).
- Можно воспользоваться файлом `application.yml` в формате YAML, который также должен располагаться в корневом каталоге пути к классам приложения (существуют и другие варианты расположения этого файла, которые я покажу вам позже).
- Можно задействовать переменные среды. Для облачных сценариев использования этот вариант все чаще применяется по умолчанию.
- И можно задействовать аргументы командной строки.

Напомню, что Spring Boot — продуманная технология. Большая часть настроек приложений Spring Boot задается в общем файле `application.properties` или `application.yml`. Если ни в одном из них ничего не задано, существуют значения этих свойств по умолчанию. Найти полный список стандартных свойств приложений можно по адресу https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html.

Одна из самых интересных возможностей Spring (и Spring Boot) — доступ к значениям свойств посредством аннотации `@Value` (с указанием названия свойства). Можно также обращаться к ним из интерфейса `org.springframework.core.env.Environment`, расширяющего интерфейс `org.springframework.core.env.PropertyResolver`. Например, при наличии файла `src/main/resources/application.properties` со свойством:

```
data.server=remoteserver:3030
```

можно обратиться в приложении к свойству `data.server` с помощью аннотации `@Value`, как показано в следующем фрагменте кода:

```
//...
@Service
public class MyService {

@Value("${data.server}")
private String server;
      //...
}
```

Этот фрагмент кода иллюстрирует использование аннотации `@Value`. Spring Boot внедряет свойство `data.server` со значением `remoteserver:3030` из файла `application.properties` в переменную `server`.

Если же вы не хотите прибегать к файлу `application.properties`, то можете внедрять свойства из командной строки.

```
$ java -jar target/myapp.jar --data.server=remoteserver:3030
```

И получите тот же результат. Если вам не нравится файл `application.properties` или вы не любите синтаксис YAML, можете воспользоваться специализированной переменной среды `SPRING_APPLICATION_JSON`, чтобы сделать доступными те же свойства и их значения.

```
$ SPRING_APPLICATION_JSON='{ "data":{"server":"remoteserver:3030"}}'
  java -jar target/myapp.jar
```

И тоже получите такой же результат (пользователям Windows необходимо сначала применить инструкцию `SET` для установки значения переменной среды). Таким образом, Spring Boot предоставляет различные варианты обеспечения доступности свойств приложений.

Примеры использования свойств конфигурации

Создадим простой проект, который поможет вам лучше разобраться с конфигурацией приложения. Откройте браузер и перейдите по адресу https://start.spring.io. Заполните поля следующими значениями.

- Group (Группа): `com.apress.spring`.
- ArtifactId (Идентификатор артефакта): `spring-boot-config`.
- Package Name (Название пакета): `com.apress.spring`.
- Name (Название): `spring-boot-config`.

Можете выбрать любой тип проекта, который вам нравится. Нажмите кнопку Generate Project (Сгенерировать проект), сохраните ZIP-файл, разархивируйте его и импортируйте в свою любимую IDE.

Прежде чем продолжить работу с проектом, учтите, что у Spring Boot существует некоторый порядок переопределения свойств конфигурации приложения:

- аргументы командной строки;
- `SPRING_APPLICATION_JSON`;

- JNDI (`java:comp/env`);
- `System.getProperties()`;
- переменные среды операционной системы;
- класс `RandomValuePropertySource` (`random.*`);
- свойства, относящиеся к конкретному профилю (`application-{профиль}.jar`) вне JAR-файла пакета;
- свойства, относящиеся к конкретному профилю (`application-{профиль}.jar`) внутри JAR-файла пакета;
- свойства приложения (файл `application.properties`) вне JAR-файла пакета;
- свойства приложения (файл `application.properties`) внутри JAR-файла пакета;
- аннотация `@PropertySource`;
- заданные с помощью метода `SpringApplication.setDefaultProperties`.

Вот такой порядок переопределения свойств конфигурации приложения. Приведем несколько примеров.

Аргументы командной строки

Перейдите в проект и отредактируйте основной класс, чтобы он выглядел так, как показано в листинге 3.17.

Листинг 3.17. src/main/java/com/apress/spring/SpringBootSimpleApplication.java

```
package com.apress.spring;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
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;

@SpringBootApplication
public class SpringBootConfigApplication {

      private static Logger log = LoggerFactory.getLogger
           (SpringBootConfigApplication.class);

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

```
    @Value("${server.ip}")
    String serverIp;

    @Bean
    CommandLineRunner values(){
        return args -> {
            log.info(" > The Server IP is: " + serverIp);
        };
    }
}
```

В листинге 3.17 приведен класс `main`. Как вы можете видеть, в нем используется аннотация `@Value("${server.ip}")`. Она преобразует текст `"${server.ip}"`, производя поиск этого свойства и его значения в упомянутом выше порядке.

Запустить данный пример можно с помощью выполнения следующей команды в корневом каталоге проекта:

```
$ ./mvnw spring-boot:run -Dserver.ip=192.168.12.1
```

Если же вы упаковали сначала приложение (создали из него исполняемый JAR-файл), то можете запустить его следующим образом:

```
$ ./mvnw package
$ java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar
    --server.ip=192.168.12.1
```

В любом случае вы увидите примерно что-то такое:

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.0.0.RELEASE)
INFO - [main] c.a.spring.SpringBootConfigApplication   : Starting
SpringBootConfigApplication v0.0..
INFO - [main] c.a.spring.SpringBootConfigApplication   : No active profile
set, falling back to de..
INFO - [main] s.c.a.AnnotationConfigApplicationContext : Refreshing org.
springframework.context.an..
INFO - [main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans
for JMX exposure on sta..
INFO - [main] c.a.spring.SpringBootConfigApplication   : Started
SpringBootConfigApplication in 0...
```

```
INFO - [main] c.a.spring.SpringBootConfigApplication   :  > The Server IP is:
192.168.34.56
INFO - [main] c.a.spring.SpringBootConfigApplication   :  > App Name: My
Config App
INFO - [main] c.a.spring.SpringBootConfigApplication   :  > App Info: This is
an example
INFO - [ T-2] s.c.a.AnnotationConfigApplicationContext : Closing org.
springframework.context.annot..
INFO - [ T-2] o.s.j.e.a.AnnotationMBeanExporter        : Unregistering JMX-
exposed beans on shutdo...
```

В выводе видно следующее: `> The Server IP is: 192.68.12.1`.

Теперь создадим файл `application.properties` (его содержимое приведено в листинге 3.18).

Листинг 3.18. Файл src/main/resources/application.properties

```
server.ip=192.168.23.4
```

Если запустить приложение с теми же аргументами командной строки, будет видно, что приоритет — у аргументов из файла `application.properties`, но запустим его без параметров, вот так:

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

или:

```
$ /.mvnw package
$ java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar
```

В этом случае мы получим текст `The Server IP is: 192.168.23.4`. Если вам часто приходится использовать формат JSON, возможно, вам захочется передавать свойства именно в данном формате. Можете воспользоваться для этого свойством `spring.application.json` и запустить приложение вот такой командой:

```
$ ./mvnw spring-boot:run -Dspring.application.json='{"server":{"
ip":"192.168.145.78"}}'
```

или:

```
$ java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar --spring.
application.json='{"server":{"ip":"192.168.145.78"}}'
```

Можно также добавить этот параметр как переменную среды.

```
$ SPRING_APPLICATION_JSON='{"server":{"ip":"192.168.145.78"}}' java -jar
target/spring-boot-config-0.0.1-SNAPSHOT.jar
```

Вы увидите текст > `The Server IP is: 192.168.145.78`. И да, добавить свою переменную среды, ссылающуюся на нужное свойство, можно следующим образом:

```
$ SERVER_IP=192.168.150.46 ./mvnw spring-boot:run
```

или:

```
$ SERVER_IP=192.168.150.46 java -jar target/spring-boot-config-0.0.1-
SNAPSHOT.jar
```

Вы увидите текст > `The Server IP is: 192.168.150.46`.

ПРИМЕЧАНИЕ

Для пользователей операционной системы Windows: не забудьте применить инструкцию SET для установки значений переменных среды.

Откуда же Spring Boot знает, что переменная среды относится к свойству `server.ip`?

Смягченная привязка

Spring Boot использует для привязки смягченные правила (табл. 3.1).

Таблица 3.1. Смягченная привязка Spring Boot

Свойство	Описание
message.destinationName	Обычный «верблюжий» регистр
message.destination-name	Нотация с использованием дефисов — рекомендуемый способ добавления свойств в файлы application.properties и YML
MESSAGE_DESTINATION_NAME	Верхний регистр — рекомендуется для переменных среды операционной системы

В табл. 3.1 приведены смягченные правила для названий свойств. Именно поэтому в предыдущем примере свойство `server.ip` распознавалось также как `SERVER_IP`. Данное смягченное правило связано с аннотацией `@ConfigurationProperties` и ее префиксом, о которых мы поговорим далее.

Изменение местоположения и названия

Spring Boot ищет файлы `application.properties` и YAML в определенном порядке.

1. В подкаталоге `/config` текущего каталога.
2. В текущем каталоге.
3. В пакете `/config`, расположенном по пути к классам.
4. В корневом каталоге пути к классам.

Можете проверить это, создав подкаталог `/config` в своем текущем каталоге, добавив в него новый файл `application.properties` и убедившись, что приведенный выше порядок соблюдается. Не забывайте, что у вас уже есть файл `application.properties` в корневом каталоге пути к классам (`src/main/resources`).

Spring Boot позволяет менять название файла свойств и его местоположение. Так, например, представьте себе, что используете каталог `/config`, а название файла свойств теперь будет `mycfg.properties` (с содержимым `server.ip=127.0.0.1`). Далее можете запустить приложение с помощью следующей команды:

```
$./mvnw spring-boot:run -Dspring.config.name=mycfg
```

или:

```
$ java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar
    --spring.config.name=mycfg
```

или:

```
$ SPRING_CONFIG_NAME=mycfg java -jar
    target/spring-boot-config-0.0.1-SNAPSHOT.jar
```

Вы должны увидеть при этом текст `> The Server IP is: 127.0.0.1`. Включать в название `.properties` не обязательно; это расширение добавляется автоматически. Можно также изменить размещение данного файла. Например, создайте подкаталог `app` и добавьте в него файл `mycfg.properties` (с содержимым `server.ip=localhost`). Теперь можете запустить приложение с помощью:

```
$ ./mvnw spring-boot:run -Dspring.config.name=mycfg
    -Dspring.config.location=file:app/
```

или:

```
$ java -jar target/spring-boot-config-0.0.1-SNAPSHOT.jar
    --spring.config.location=file:app/ --spring.config.name=mycfg
```

Или можете добавить файл `mycfg.properties` в каталог `src/main/resources/META-INF/conf` (можете его создать) и выполните следующее:

```
$ mkdir -p src/main/resources/META-INF/conf
$ cp config/mycfg.properties src/main/resources/META-INF/conf/
$ ./mvnw clean spring-boot:run -Dspring.config.name=mycfg -Dspring.config.
location=classpath:META-INF/conf/
```

Вы должны увидеть текст `> The Server IP is: 127.0.0.1`.

Попробуйте изменить это значение и убедиться, что Spring Boot действительно производит поиск по пути к классам.

Spring Boot также придерживается определенного порядка при поиске файла свойств.

1. Путь к классам (`classpath`).
2. `classpath:/config`.
3. `file:`.
4. `file:config/`.

Если не поменять ее с помощью свойства `spring.config.location`, то для изменения местоположения файла свойств используется переменная среды `SPRING_CONFIG_LOCATION`.

Свойства, относящиеся к конкретному профилю

Начиная с версии 3.1, если я не путаю, в фреймворке Spring появилась замечательная возможность, позволяющая разработчику создавать пользовательские свойства и компоненты, относящиеся к конкретным профилям. Это удобный способ разделения сред без необходимости заново компилировать или упаковывать приложение Spring. Все, что нужно сделать, — задать активный профиль с помощью аннотации `@ActiveProfiles` или получить текущий объект `Environment`[1] и воспользоваться методом `setActiveProfiles`. Или можно применить переменную среды `SPRING_PROFILES_ACTIVE` либо свойство `spring.profiles.active`.

Можно также указать файл свойств в следующем формате: `application-{профиль}.properties`. Создадим два файла в подкаталоге `config/`: `application-`

[1] Точнее, объект типа ConfigurableEnvironment.

qa.properties и application-prod.properties. Взглянем на содержимое каждого из них.

- Файл application-qa.properties:

```
server.ip=localhost
```

- Файл application-prod.properties:

```
server.ip=http://my-remote.server.com
```

Теперь можно запустить наш пример с помощью команды:

```
$ ./mvnw clean spring-boot:run -Dspring.profiles.active=prod
```

При выполнении этой команды взгляните на начало журнала. Вы должны увидеть нечто схожее со следующим выводом:

```
...
INFO 58270 --- [main] c.a.spring... : The following profiles are active: prod
...
INFO 58270 --- [main] c.a.spring... : > The Server IP is: http://my-remote.
server.com
INFO 58270 --- [main] c.a.spring... : > App Name: Super App
INFO 58270 --- [main] c.a.spring... : > App Info: This is production
```

Вы увидите в консоли надпись The following profiles are active: prod и, конечно, активное значение свойств приложения (application-prod.properties): > The Server IP is: http://my-remote.server.com. В качестве упражнения попробуйте изменить название application-prod.properties на mycfg-qa.properties и использовать свойства Spring с новым названием. Если не задать никаких активных профилей, будет использоваться профиль по умолчанию, то есть файл application.properties.

Пользовательский префикс для свойств

Spring Boot дает возможность указывать и использовать пользовательский префикс для ваших свойств. Необходимо только снабдить аннотацией @ConfigurationProperties Java-класс с сеттерами и геттерами для свойств.

Если вы применяете IDE STS, рекомендую включить следующую зависимость в файл pom.xml или build.gradle (в зависимости от используемой системы управления зависимостями). Эта зависимость создает code-insight

(понимание кода) и запускает в редакторе автодополнение для свойств. Так, если вы используете Maven, то можете добавить в файл `pom.xml` следующую зависимость:

```
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-configuration-processor</artifactId>
      <optional>true</optional>
</dependency>
```

Если вы используете Gradle, то можете добавить в файл `build.gradle` такую зависимость:

```
dependencies {
      optional "org.springframework.boot:spring-boot-configuration-processor"
}
compileJava.dependsOn(processResources)
```

Эта зависимость дает возможность обработки пользовательских свойств и обеспечивает автодополнение кода. Теперь рассмотрим пример. Измените файл `src/main/resource/application.properties` так, чтобы он выглядел, как показано в листинге 3.19.

Листинг 3.19. Файл src/main/resources/application.properties

```
server.ip=192.168.3.5

myapp.server-ip=192.168.34.56
myapp.name=My Config App
myapp.description=This is an example
```

В листинге 3.19 приведен файл `src/main/resource/application.properties`. Второй блок в нем — новый. В нем описаны пользовательские свойства с префиксом `myapp`. Откройте основной класс своего приложения и отредактируйте его в соответствии с листингом 3.20.

Листинг 3.20. src/main/java/com/apress/spring/SpringBootSimpleApplication.java, версия 9

```
package com.apress.spring;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
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.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.stereotype.Component;

@SpringBootApplication
public class SpringBootConfigApplication {

      private static Logger log = LoggerFactory.getLogger
           (SpringBootConfigApplication.class);

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

      @Value("${myapp.server-ip}")
      String serverIp;

      @Autowired
      MyAppProperties props;

      @Bean
      CommandLineRunner values(){
            return args -> {
                  log.info(" > The Server IP is: " + serverIp);
                  log.info(" > App Name: " + props.getName());
                  log.info(" > App Info: " + props.getDescription());
            };
      }

      @Component
      @ConfigurationProperties(prefix="myapp")
      public static class MyAppProperties {
            private String name;
            private String description;
            private String serverIp;

            public String getName() {
                  return name;
            }

            public void setName(String name) {
                  this.name = name;
            }
            public String getDescription() {
                  return description;
            }
            public void setDescription(String description) {
                  this.description = description;
            }
            public String getServerIp() {
                  return serverIp;
            }
```

```
        public void setServerIp(String serverIp) {
            this.serverIp = serverIp;
        }
    }
}
```

В листинге 3.19 показан основной класс приложения. Изучим его детальнее.

- `@Value("${myapp.server-ip}")`. Теперь в аннотации указано `myapp.server-ip`, то есть значение будет равно 192.68.34.56.
- `@Autowired MyAppProperties props`. Этот код создает экземпляр типа `MyAppProperties`.
- `@Component @ConfigurationProperties(prefix="myapp")`. Аннотация `@ConfigurationProperties` сообщает Spring Boot, что класс используется для всех описанных в файле `application.properties` аннотаций с префиксом `myapp`. Это значит, что он распознает, например, свойства `myapp.serverIp` (или `myapp.server-ip`), `myapp.name` и `myapp.description`. Аннотация `@Component` просто отмечает класс как кандидат в компоненты для последующего автоматического обнаружения.

Spring Boot использует смягченные правила привязки свойств `Environment` с компонентами `@ConfigurationProperties`, так что никакие конфликты названий не возможны.

Теперь, если запустить приложение, мы увидим все свойства `myapp`.

```
$ ./mvnw clean spring-boot:run
...
> The Server IP is: 192.168.34.56
> App Name: My Config App
> App Info: This is an example
...
```

Как видите, существует множество возможных вариантов применения свойств конфигурации приложения. Если вы хотите использовать синтаксис YAML, пожалуйста, обратитесь к документации Spring Boot за примерами.

Вы можете включить в своем IDE дополнительные подсказки (`hints`), предоставив ему информацию для автодополнения свойств. Узнать, как создавать эти метаданные, можно из справочной документации по веб-адресу https://docs.spring.io/spring-boot/docs/current/reference/html/configuration-metadata.html.

Резюме

В этой главе вам предоставлен обзор внутреннего устройства Spring Boot в виде рассказа о возможности автоконфигурации, включая «закулисные» детали функционирования аннотации `@EnableAutoConfiguration`. Вы также научились исключать отдельные из автоконфигурационных классов.

Вы узнали о некоторых из возможностей Spring Boot и способов использования свойств конфигурации приложения, а также научились подстраивать свойства конфигурации приложения под свои нужды посредством добавления префикса.

В следующей главе вы узнаете больше о создании веб-приложений с помощью Spring Boot.

4 Создание веб-приложений

Сегодня Интернет — основной канал обмена информацией для любых приложений: от предназначенных для настольных компьютеров и до мобильных устройств, от приложений соцсетей и коммерческих приложений до игр и от простого контента до потоковых данных. А Spring Boot может помочь в разработке веб-приложений нового поколения.

В этой главе рассказывается, как легко и просто создавать веб-приложения на Spring Boot. Вы уже знаете из примеров в предыдущих главах, что можно делать с веб-приложениями. Вы знаете, что Spring Boot упрощает создание веб-приложения, сводя его к нескольким строкам кода, и позволяет не думать о файлах конфигурации и не искать сервер приложений для развертывания веб-приложения. Благодаря Spring Boot и его автоконфигурациям можно использовать встроенный сервер приложений, например Tomcat, Netty, Undertow или Jetty, что делает ваше приложение легко распространяемым и переносимым.

Spring MVC

Начнем с обсуждения технологии Spring MVC и некоторых ее возможностей. Напомню, что фреймворк Spring состоит примерно из 20 модулей (технологий), одна из которых — веб-технология. Для работы с веб-технологиями во фреймворке Spring предназначены модули `spring-web`, `spring-webmvc`, `spring-webflux` и `spring-websocket`.

Модуль `spring-web` включает основные возможности веб-интеграции, например функциональность загрузки файлов по частям, инициализацию

контейнера Spring (с помощью сервлет-прослушивателей) и класс контекста приложения, ориентированный на работу с Интернетом. Модуль `spring-webmvc` (модуль веб-сервера) содержит реализации всех сервисов Spring MVC (Model — View — Controller, «Модель — Представление — Контроллер») и REST для веб-приложений. Эти модули включают множество возможностей, в частности библиотеки JSP-тегов с очень широкой функциональностью, настраиваемую привязку и проверку корректности данных, гибкий перенос модели, настраиваемое разрешение обработчиков и представлений и т. д.

В основе архитектуры Spring MVC лежит `org.springframework.web.servlet.DispatcherServlet`. Этот сервлет очень гибок и отличается отказоустойчивой функциональностью, которую не найти ни в одном другом веб-фреймворке MVC. `DispatcherServlet` предоставляет несколько готовых стратегий разрешения неоднозначностей, включая разрешение представлений, региональных настроек, разрешение тем и обработчики исключений. Другими словами, `DispatcherServlet` перенаправляет полученный HTTP-запрос нужному обработчику (классу, помеченному аннотацией `@Controller` или `@RestController`, и методам, использующим аннотации `@RequestMapping`) и правильному представлению (JSP-сценарию).

Автоконфигурация Spring Boot MVC

Можно легко создавать веб-приложения путем добавления зависимости `spring-boot-starter-web` в файл `pom.xml` или `build.gradle`. Такая зависимость обеспечивает все нужные JAR-файлы `spring-web` и некоторые дополнительные, например `tomcat-embed*` и `jackson` (для JSON и XML). Это значит, что Spring Boot использует все возможности модулей Spring MVC и обеспечивает всю необходимую автоконфигурацию для создания нужной веб-инфраструктуры, в частности настройки `DispatcherServlet`, предоставляет значения по умолчанию (если вы их не переопределите), настраивает встроенный сервер Tomcat (чтобы можно было запускать приложение без каких-либо контейнеров) и многое другое.

Автоконфигурация добавляет в веб-приложение следующие возможности.

- *Поддержка статического контента*, то есть возможность добавления статического контента, например HTML, JavaScript, CSS, мультимедийных элементов и т. д., в каталог `/static` (по умолчанию) или `/public`, `/resources` или `/META-INF/resources`, который должен находиться по пути к классам или в текущем каталоге. Spring Boot подхватывает их и выдает по запросу.

Это поведение можно легко менять, модифицируя свойство `spring.mvc.static-path-pattern` или `spring.resources.static-locations`. Одна из замечательных возможностей Spring Boot относительно веб-приложений: если создать файл `index.html`, Spring Boot выдаст его автоматически без регистрации компонентов или потребности в дополнительной конфигурации.

- *Компонент* `HttpMessageConverters`. При необходимости получить JSON-ответ в случае обычного приложения Spring MVC следует создать соответствующую конфигурацию (XML или JavaConfig) для компонента `HttpMessageConverters`. Spring Boot добавляет данную поддержку по умолчанию, так что этого делать не нужно; это значит, что формат JSON выдается по умолчанию (благодаря библиотекам `jackson`, предоставляемым модулем `spring-boot-starter-web` в виде зависимостей). А если автоконфигурация Spring Boot обнаруживает XML-расширение Jackson по пути к классам, то агрегирует `HttpMessageConverter` для XML с прочими средствами преобразования, в результате чего приложение может выдать контент в формате `application/json` или `application/xml`, в зависимости от заголовка `content-type` запроса.
- *Сериализаторы и десериализаторы JSON.* Чтобы лучше управлять сериализацией/десериализацией в/из формата JSON, Spring Boot предоставляет возможность создания своих собственных сериализаторов/десериализаторов путем расширения типов `JsonSerializer<T>` и/или `JsonDeserializer<T>` и снабжения созданного класса аннотацией `@JsonComponent`, чтобы его можно было зарегистрировать для использования. Еще одна возможность Spring Boot — поддержка Jackson; по умолчанию Spring Boot сериализует поля дат в формате `2018-05-01T23:31:38.141+0000`, но такое поведение по умолчанию можно поменять, изменив значение свойства `spring.jackson.date-format=yyyy-MM-dd` (можно использовать любой паттерн формата даты); предыдущее значение приводит к генерации вывода вида `2018-05-01`.
- *Сопоставление с путем и согласование контента.* Одна из принятых практик работы приложений Spring MVC — способность реагировать на любой суффикс для представления *типа контента ответа* и согласования его контента. Например, в случае `/api/todo.json` или `/api/todo.pdf` `content-type` будет `application/json` и `application/pdf`; так что ответ будет в формате JSON или представлять собой PDF-файл соответственно. Другими словами, Spring MVC производит сопоставление с паттерном `.*` суффикса, например `/api/todo.*`. Spring Boot отключает такое поведение по умолчанию. При этом все равно можно применять для данной цели параметр, установив свойство `spring.mvc.contentnegotiation.favor-parameter=true`

(по умолчанию `false`); так что можно использовать что-то вроде `/api/todo?format=xml` (`format` — название этого параметра по умолчанию; конечно, его можно поменять, задав `spring.mvc.contentnegotiation.parameter-name=myparam`). В результате `content-type` становится `application/xml`.

- *Обработка ошибок.* Spring Boot использует путь `/error` для создания страницы, на которой выводятся все глобальные ошибки. Такое поведение можно поменять, создав свои собственные пользовательские страницы для этой цели. Необходимо создать пользовательские HTML-страницы в каталоге `src/main/resources/public/error/`, например страницы `500.html` или `404.html`. В случае воплощающего REST приложения Spring Boot отвечает в формате JSON. Spring Boot также поддерживает обработку ошибок с помощью Spring MVC при применении аннотаций `@ControllerAdvice` или `@ExceptionHandler`. Зарегистрировать пользовательские объекты `ErrorPage` можно путем реализации интерфейса `ErrorPageRegistrar` и объявления его как компонента Spring.
- *Поддержка шаблонизаторов.* Spring Boot поддерживает FreeMarker, Groovy Templates, Thymeleaf и Mustache. При включении зависимости `spring-boot-starter-<шаблонизатор>` необходима автоконфигурация Spring Boot для ее активации и добавления всех нужных средств разрешения представлений и обработчиков файлов. По умолчанию Spring Boot ищет их в каталоге `src/main/resources/templates/path`.

Автоконфигурация Spring Boot Web предоставляет и множество других возможностей. Пока что нас интересует только технология сервлетов, но очень скоро мы займемся новейшим прибавлением в семействе Spring Boot: *WebFlux*.

Spring Boot Web: приложение ToDo

Чтобы лучше разобраться в работе Spring Boot с веб-приложениями и возможностями модулей Spring MVC, вам предстоит создать приложение ToDo, которое предоставляет REST API. Вот список требований:

- создайте модель предметной области ToDo со следующими полями и типами: `id` (строковое значение), `description` (строковое значение), `completed` (булево значение), `created` (дата и время), `modified` (дата и время);
- создайте REST API, обеспечивающий основные действия CRUD (создание, чтение, обновление и удаление). Используйте самые распространенные методы HTTP: `POST`, `PUT`, `PATCH`, `GET` и `DELETE`;

- создайте репозиторий для хранения состояния списка из нескольких запланированных дел (ToDo). Пока достаточно будет репозитория, размещаемого в оперативной памяти;
- добавьте обработчик ошибок на случай испорченного запроса или отсутствия у отправляемого нового ToDo всех требуемых полей. Единственное обязательное поле — `description`;
- все запросы и ответы должны быть в формате JSON.

Приложение ToDo

Откройте браузер и перейдите по адресу https://start.spring.io, чтобы создать приложение ToDo на основе следующих значений (рис. 4.1).

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-in-memory`.
- Name (Название): `todo-in-memory`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Web`, `Lombok`.

Благодаря выбору зависимости `Lombok` можно с легкостью создать классы модели предметной области и избавляет от написания стереотипных сеттеров, геттеров и других переопределений.

ПРИМЕЧАНИЕ

Больше информации о Lombok можно найти в справочной документации по адресу https://projectlombok.org.

В качестве типа проекта можете выбрать Maven или Gradle; в этой книге мы используем и тот и другой. Нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами IDE. Вот некоторые из лучших IDE: STS (https://spring.io/tools/sts/all), IntelliJ IDEA (www.jetbrains.com/idea/) и VSCode (https://code.visualstudio.com/). Я рекомендую воспользоваться одной из этих IDE ради возможности автодополнения кода, благодаря которой видно, какие методы или параметры можно в него добавить.

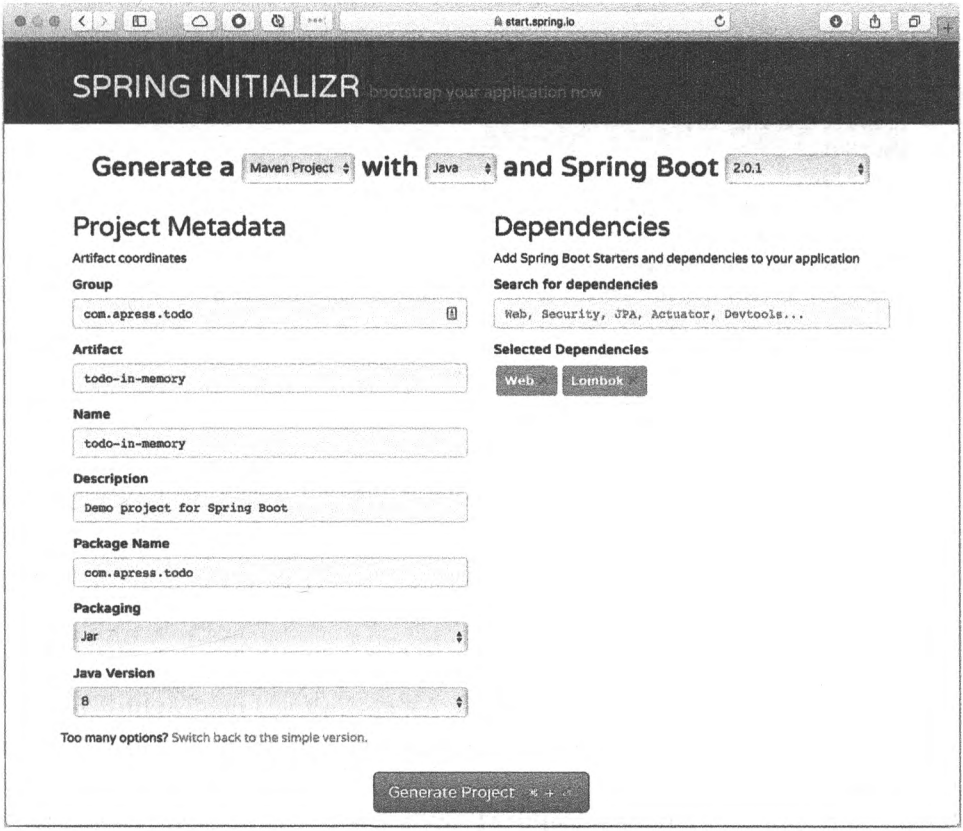


Рис. 4.1. Создание приложения ToDo на странице https://start.spring.io

Модель предметной области: класс ToDo

Согласно требованиям сначала нужно создать класс `ToDo` для модели предметной области (листинг 4.1).

Листинг 4.1. com.apress.todo.domain.ToDo.java

```
package com.apress.todo.domain;

import lombok.Data;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;
import java.util.UUID;
```

```
@Data
public class ToDo {

    @NotNull
    private String id;
    @NotNull
    @NotBlank
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
        LocalDateTime date = LocalDateTime.now();
        this.id = UUID.randomUUID().toString();
        this.created = date;
        this.modified = date;
    }

    public ToDo(String description){
        this();
        this.description = description;
    }
}
```

В листинге 4.1 приведен класс `ToDo`, включающий все необходимые поля. Он также снабжен аннотацией `@Data` — аннотацией `Lombok`, которая генерирует конструктор по умолчанию (если у вас его нет) и все сеттеры, геттеры и переопределения методов, например метода `toString`, чтобы сделать класс чище. Отметьте также, что для некоторых полей в классе указаны аннотации `@NotNull` и `@NotBlank`; они используются при предстоящей нам в дальнейшем проверке корректности данных. Конструктор по умолчанию способен задавать начальные значения полей для удобства создания экземпляра `ToDo`.

Текучий API: класс ToDoBuilder

Теперь мы создадим класс текучего API для упрощения создания экземпляра `ToDo`. Как видите, этот класс представляет собой фабрику, создающую экземпляр `ToDo` вместе с описанием или конкретным ID (листинг 4.2).

Листинг 4.2. com.apress.todo.domain.ToDoBuilder.java

```
package com.apress.todo.domain;

public class ToDoBuilder {
```

```
    private static ToDoBuilder instance = new ToDoBuilder();
    private String id = null;
    private String description = "";

    private ToDoBuilder(){}

    public static ToDoBuilder create() {
        return instance;
    }

    public ToDoBuilder withDescription(String description){
        this.description = description;
        return instance;
    }

    public ToDoBuilder withId(String id){
        this.id = id;
        return instance;
    }

    public ToDo build(){
        ToDo result = new ToDo(this.description);
        if(id != null)
            result.setId(id);
        return result;
    }
}
```

Листинг 4.2 представляет собой простой класс-фабрику для создания экземпляра класса `ToDo`. Мы расширим его функциональность в следующих главах.

Репозиторий: интерфейс CommonRepository<T>

Создадим интерфейс, включающий стандартные действия по сохранению данных. Этот интерфейс — параметризованный, что упрощает использование различных реализаций, благодаря чему наш репозиторий представляет собой легко расширяемое решение (листинг 4.3).

Листинг 4.3. com.apress.todo.repository.CommonRepository<T>.java

```
package com.apress.todo.repository;

import java.util.Collection;

public interface  CommonRepository<T> {
    public T save(T domain);
    public Iterable<T> save(Collection<T> domains);
```

```
    public void delete(T domain);
    public T findById(String id);
    public Iterable<T> findAll();
}
```

Листинг 4.3 представляет собой общий интерфейс, который можно использовать для любой реализации постоянного хранения данных. Конечно, эти сигнатуры можно изменить в любой момент. Это просто пример создания расширяемого решения.

Репозиторий: класс ToDoRepository

Теперь создадим конкретный класс, реализующий интерфейс `CommonRepository<T>`. Напомню спецификацию: пока что нужно только хранить список запланированных дел в оперативной памяти (листинг 4.4).

Листинг 4.4. com.apress.todo.repository.ToDoRepository.java

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.stereotype.Repository;

import java.time.LocalDateTime;
import java.util.Collection;
import java.util.Comparator;
import java.util.HashMap;
import java.util.Map;
import java.util.stream.Collectors;

@Repository
public class ToDoRepository implements CommonRepository<ToDo> {

    private Map<String,ToDo> toDos = new HashMap<>();

    @Override
    public ToDo save(ToDo domain) {
        ToDo result = toDos.get(domain.getId());
        if(result != null) {
            result.setModified(LocalDateTime.now());
            result.setDescription(domain.getDescription());
            result.setCompleted(domain.isCompleted());
            domain = result;
        }
        toDos.put(domain.getId(), domain);
        return toDos.get(domain.getId());
    }
```

```
    @Override
    public Iterable<ToDo> save(Collection<ToDo> domains) {
        domains.forEach(this::save);
        return findAll();
    }

    @Override
    public void delete(ToDo domain) {
        toDos.remove(domain.getId());
    }

    @Override
    public ToDo findById(String id) {
        return toDos.get(id);
    }

    @Override
    public Iterable<ToDo> findAll() {
         return toDos.entrySet().stream().sorted(entryComparator).map
             (Map.Entry::getValue).collect(Collectors.toList());
    }

    private Comparator<Map.Entry<String,ToDo>> entryComparator =
         (Map.Entry<String, ToDo> o1, Map.Entry<String, ToDo> o2) -> {
         return o1.getValue().getCreated().compareTo
             (o2.getValue().getCreated());
    };
}
```

В листинге 4.4 приведена реализация интерфейса `CommonRepository<T>`. Просмотрите и проанализируйте его код. Для хранения всех объектов `ToDo` в нем используется хеш-карта. Благодаря самой природе хеш-карты все операции упрощаются, что облегчает реализацию.

Проверка корректности данных: класс ToDoValidationError

Теперь создадим класс для проверки корректности данных, выявляющий все возможные ошибки при работе приложения, например запланированное дело без описания. Напомню, что в классе `ToDo` поля `ID` и `description` помечены аннотацией `@NotNull`. А поле описания снабжено еще и дополнительной аннотацией `@NotBlank` для гарантии, что оно никогда не будет пустым[1] (листинг 4.5).

[1] Существует также предназначенная для аналогичной цели аннотация NotEmpty: https://javaee.github.io/javaee-spec/javadocs/javax/validation/constraints/package-summary.html.

Листинг 4.5. com.apress.todo.validation.ToDoValidationError.java

```
package com.apress.todo.validation;

import com.fasterxml.jackson.annotation.JsonInclude;

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

public class ToDoValidationError {

    @JsonInclude(JsonInclude.Include.NON_EMPTY)
    private List<String> errors = new ArrayList<>();

    private final String errorMessage;

    public ToDoValidationError(String errorMessage) {
        this.errorMessage = errorMessage;
    }

    public void addValidationError(String error) {
        errors.add(error);
    }

    public List<String> getErrors() {
        return errors;
    }

    public String getErrorMessage() {
        return errorMessage;
    }
}
```

В листинге 4.5 показан класс `ToDoValidationError`, предназначенный для хранения возникающих при запросах ошибок. Он использует дополнительную аннотацию `@JsonInclude`, означающую, что поле `errors` должно быть включено, даже если оно пусто.

Проверка корректности данных: фабрика ToDoValidationErrorBuilder

Создадим еще одну фабрику для получения экземпляров класса `ToDoValidationError` (листинг 4.6).

Листинг 4.6. com.apress.todo.validation.ToDoValidationErrorBuilder.java

```
package com.apress.todo.validation;

import org.springframework.validation.Errors;
import org.springframework.validation.ObjectError;
```

```
public class ToDoValidationErrorBuilder {

    public static ToDoValidationError fromBindingErrors(Errors errors) {
        ToDoValidationError error = new ToDoValidationError("Validation
            failed. " + errors.getErrorCount() + " error(s)");
        for (ObjectError objectError : errors.getAllErrors()) {
            error.addValidationError(objectError.getDefaultMessage());
        }
        return error;
    }
}
```

Листинг 4.6 демонстрирует еще один класс-фабрику, который с легкостью создает экземпляр `ToDoValidationError` со всей необходимой информацией.

Контроллер: класс ToDoController

Пришло время создать REST API, используя все приведенные выше классы. Вам предстоит создать класс `ToDoController`, в котором будут видны все возможности Spring MVC, аннотации, настройка конечных точек и обработка ошибок.

Взгляните на приведенный в листинге 4.7 код.

Листинг 4.7. com.apress.todo.controller.ToDoController.java

```
package com.apress.todo.controller;

import com.apress.todo.domain.ToDo;
import com.apress.todo.domain.ToDoBuilder;
import com.apress.todo.repository.CommonRepository;
import com.apress.todo.validation.ToDoValidationError;
import com.apress.todo.validation.ToDoValidationErrorBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import javax.validation.Valid;
import java.net.URI;

@RestController
@RequestMapping("/api")
public class ToDoController {

    private CommonRepository<ToDo> repository;
```

```
@Autowired
public ToDoController(CommonRepository<ToDo> repository) {
    this. repository = repository;
}

@GetMapping("/todo")
public ResponseEntity<Iterable<ToDo>> getToDos(){
    return ResponseEntity.ok(repository.findAll());
}

@GetMapping("/todo/{id}")
public ResponseEntity<ToDo> getToDoById(@PathVariable String id){
    return ResponseEntity.ok(repository.findById(id));
}

@PatchMapping("/todo/{id}")
public ResponseEntity<ToDo> setCompleted(@PathVariable String id){
    ToDo result = repository.findById(id);
    result.setCompleted(true);
    repository.save(result);
    URI location = ServletUriComponentsBuilder.fromCurrentRequest()
            .buildAndExpand(result.getId()).toUri();

    return ResponseEntity.ok().header
         ("Location",location.toString()).build();
}

 @RequestMapping(value="/todo", method = {RequestMethod.POST,
 RequestMethod.PUT})
 public ResponseEntity<?> createToDo(@Valid @RequestBody ToDo toDo,
                                     Errors errors){
    if (errors.hasErrors()) {
         return ResponseEntity.badRequest().
             body(ToDoValidationErrorBuilder.fromBindingErrors(errors));
    }

    ToDo result = repository.save(toDo);
    URI location = ServletUriComponentsBuilder.fromCurrentRequest().
        path("/{id}").buildAndExpand(result.getId()).toUri();
    return ResponseEntity.created(location).build();
}

@DeleteMapping("/todo/{id}")
public ResponseEntity<ToDo> deleteToDo(@PathVariable String id){
    repository.delete(ToDoBuilder.create().withId(id).build());
    return ResponseEntity.noContent().build();
}
```

```
    @DeleteMapping("/todo")
    public ResponseEntity<ToDo> deleteToDo(@RequestBody ToDo toDo){
        repository.delete(toDo);
        return ResponseEntity.noContent().build();
    }

  @ExceptionHandler
    @ResponseStatus(value = HttpStatus.BAD_REQUEST)
    public ToDoValidationError handleException(Exception exception) {
        return new ToDoValidationError(exception.getMessage());
    }
}
```

В листинге 4.7 приведен класс `ToDoController`. Изучим его подробнее.

- Аннотация `@RestController`. Spring MVC дает возможность объявлять отображение запросов, входные данные запросов, обработку ошибок и др. с помощью аннотаций `@Controller` и `@RestController`. Всю функциональность берут на себя эти аннотации, так что расширять или реализовывать для этого специальные интерфейсы не нужно.
- Аннотация `@RequestMapping`. Эта аннотация задает отображение запросов на методы контроллера. У нее есть несколько атрибутов для URL, HTTP-методов (`GET`, `PUT`, `DELETE` и т. д.), параметров запроса, заголовков и типов мультимедийных элементов. Ее можно использовать на уровне класса (чтобы применять отображения совместно) или метода для отображения конкретных конечных точек. В данном случае мы указали атрибут `"/api"`, означающий, что у всех методов будет данный префикс.
- Аннотация `@Autowired`. Контроллер снабжен аннотацией `@Autowired`, то есть он внедряет реализацию `CommonRepository<ToDo>`. Эту аннотацию можно опустить; Spring автоматически внедряет все объявленные зависимости, начиная с версии 4.3.
- Аннотация `@GetMapping`. Это сокращенное написание аннотации `@RequestMapping`, предназначенное для HTTP-метода `GET`. `@GetMapping` эквивалентна `@RequestMapping(value="/todo", method = {RequestMethod.GET})`.
- Аннотация `@PatchMapping`. Это сокращенное написание аннотации `@RequestMapping`; в данном классе она помечает запланированное дело как выполненное.
- Аннотация `@DeleteMapping`. Это сокращенное написание аннотации `@RequestMapping`; используется для удаления запланированного дела. Наш класс

включает два перегруженных метода `deleteToDo`: один принимает на входе строковое значение, а второй — экземпляр `ToDo`.

- Аннотация `@PathVariable`. Эта аннотация удобна при объявлении конечной точки, содержащей паттерн URL-выражения; в данном случае `"/api/todo/{id}"`, в котором `id` должен соответствовать названию параметра метода.
- Аннотация `@RequestBody`. Эта аннотация отправляет запрос, включающий тело. В обычном случае, при отправке данных формы или конкретного контента, этот класс получает объект `ToDo` в формате JSON, после чего `HttpMessageConverter` десериализует JSON в экземпляр `ToDo`; это происходит автоматически благодаря фреймворку Spring Boot и его автоконфигурации, который регистрирует `MappingJackson2HttpMessageConverter` по умолчанию.
- Класс `ResponseEntity<T>`. Этот класс возвращает полный ответ, включая HTTP-заголовки, причем тело преобразовывается с помощью `HttpMessageConverter` и записывается в HTTP-ответ. Класс `ResponseEntity<T>` поддерживает текучий API, благодаря чему создание ответа не представляет сложностей.
- Аннотация `@ResponseStatus`. Обычно эта аннотация используется для методов с возвращаемым типом `void` (или пустым возвращаемым значением). Она отправляет обратно заданный в ответе код состояния HTTP.
- Аннотация `@Valid`. Эта аннотация проверяет корректность поступающих данных и используется в параметрах метода. Для запуска проверки данные, которые требуется проверить, должны быть снабжены какой-либо из аннотаций — `@NotNull`, `@NotBlank` и т. п. В этом случае в классе `ToDo` указанными аннотациями снабжены поля для идентификатора и описания. В случае нахождения ошибок при проверке они собираются в классе `Errors` (в данном случае в качестве глобального средства проверки регистрируется и используется *hibernate validator*, поставляемый вместе с JAR-файлами модуля `spring-webmvc`; но вы можете создать свое собственное пользовательское средство проверки и переопределить параметры Spring Boot по умолчанию). Далее можно изучить найденное и добавить необходимую логику для отправки сообщения об ошибке.
- Аннотация `@ExceptionHandler`. Spring MVC автоматически объявляет встроенные средства разрешения для исключений и добавляет поддержку в этой аннотации. В данном случае внутри класса контроллера объявлен метод с аннотацией `@ExceptionHandler` (ее можно использовать также и внутри перехватчика `@ControllerAdvice`) и все исключения направляются методу

handleException. При необходимости можно и конкретизировать, какие исключения обрабатывать. Например, можно организовать обработку с помощью этого метода исключения DataAccessException.

```
@ExceptionHandler
@ResponseStatus(value = HttpStatus.BAD_REQUEST)
public ToDoValidationError
   handleException(DataAccessException exception) {
                  return new
        ToDoValidationError(exception.getMessage());
}
```

Один из методов класса принимает два HTTP-метода: POST и PUT. Аннотация @RequestMapping может принимать несколько HTTP-методов, так что можно легко назначить для их обработки один метод (например, @RequestMapping(value="/todo", method = {RequestMethod.POST, RequestMethod.PUT}).

Мы охватили все необходимые для этого приложения требования, осталось его запустить и посмотреть на результаты.

Запуск: приложение ToDo

Теперь мы готовы запустить приложение ToDo и проверить, как оно работает. Если вы используете IDE (STS или IntelliJ), можете щелкнуть правой кнопкой мыши на главном классе приложения (TodoInMemoryApplication.java) и выбрать Run Action. Если же в вашем редакторе таких возможностей нет, можете открыть окно терминала и выполнить команды из листинга 4.8 или 4.9.

Листинг 4.8. В случае типа проекта Maven

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

Листинг 4.9. В случае типа проекта Gradle

```
./gradlew spring-boot:run
```

Spring Initializr (https://start.spring.io) всегда предоставляет адаптеры для выбранного типа проекта (адаптеры Maven или Gradle), так что заранее устанавливать Maven или Gradle необходимости нет.

Spring Boot по умолчанию настраивает для веб-приложений встроенный сервер Tomcat, так что можно легко запустить приложение без развертывания его в контейнере сервлетов. По умолчанию используется порт 8080.

Тестирование: приложение ToDo

Тестирование приложения ToDo не должно вызвать никаких сложностей. Выполнить его можно с помощью команд терминала или специально созданного клиентского приложения. Если вас интересует модульное или интеграционное тестирование — о них мы поговорим в одной из других глав. А здесь мы воспользуемся командой cURL. В любой Unix-системе эта команда присутствует по умолчанию, а если вы работаете с операционной системой Windows[1], то можете скачать ее с сайта https://curl.haxx.se/download.html.

При первом запуске в нашем приложении ToDo никаких запланированных дел еще не должно быть. Убедиться в этом можно с помощью следующей команды в отдельном окне терминала:

```
curl -i http://localhost:8080/api/todo
```

При этом вы должны увидеть примерно следующее:

```
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 02 May 2018 22:10:19 GMT
[]
```

Нас интересует конечная точка `/api/todo`, и, если взглянуть на листинг 4.7, видно, что метод `getToDos` возвращает `ResponseEntity<Iterable<ToDo>>`, коллекцию объектов `ToDo`. По умолчанию ответ возвращается в формате JSON (см. заголовок `Content-Type`). В ответе отправляются HTTP-заголовки и состояние.

Добавим несколько запланированных дел с помощью следующей команды[2].

```
curl -i -X POST -H "Content-Type: application/json" -d'{"description":"Read
the Pro Spring Boot 2nd Edition Book"}' http://localhost:8080/api/todo
```

---

[1] Начиная с Windows 10 v1803, операционная система уже поставляется с копией CURL. Она уже настроена, и вы можете сразу приступать к ее использованию.

[2] Не исключено, что в Windows вы столкнетесь с проблемой из-за одинарных кавычек. В этом случае можно (здесь и далее) воспользоваться вместо них экранированными двойными, например вот так:

curl -i -X POST -H "Content-Type: application/json" -d \"{"description":"Read the Pro Spring Boot 2nd Edition Book"}\» http://localhost:8080/api/todo

или можно сохранить JSON в файл (скажем, json.txt) и указать этот файл в команде:

curl -i -X POST -H "Content-Type: application/json" -d @json.txt http://localhost:8080/api/todo

В этой команде используются HTTP-метод `POST` (`-X POST`) и данные (`-d`) в формате JSON. Мы отправляем только поле описания (`description`). Важно задать заголовок (`-H`) с правильным типом контента и указать на конечную точку `/api/todo`. После выполнения этой команды вы должны увидеть примерно следующее:

```
HTTP/1.1 201
Location: http://localhost:8080/api/todo/d8d37c51-10a8-4c82-a7b1-
b72b5301cdab
Content-Length: 0
Date: Wed, 02 May 2018 22:21:09 GMT
```

Был возвращен заголовок местоположения, по которому можно прочитать запланированное дело. В `Location` виден идентификатор только что созданного объекта `ToDo`. Сгенерирован этот ответ был методом `createToDo`. Добавьте еще хотя бы два `ToDo`, чтобы данных было побольше.

Если выполнить первую из приведенных команд еще раз для получения всех запланированных дел, вы увидите примерно следующее:

```
curl -i http://localhost:8080/api/todo
HTTP/1.1 200
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 02 May 2018 22:30:16 GMT

[{"id":"d8d37c51-10a8-4c82-a7b1-b72b5301cdab","description":"Read the
 Pro Spring Boot 2nd Edition Book","created":"2018-05-02T22:27:26.042+0
 000","modified":"2018-05-02T22:27:26.042+0000","completed":false},{"id"
 :"fbb20090-19f5-4abc-a8a9-92718c2c4759","description":"Bring Milk after
 work","created":"2018-05-02T22:28:23.249+0000","modified":"2018-05-02T22:
 28:23.249+0000","completed":false},{"id":"2d051b67-7716-4ee6-9c45-1de939-
 fa579f","description":"Take the dog for a walk","created":"2018-05-02T22:29
 :28.319+0000","modified":"2018-05-02T22:29:28.319+0000","completed":false}]
```

Конечно, данные выведены не в самом красивом виде, но мы получили весь список запланированных дел. Можно отформатировать этот вывод с помощью еще одной утилиты командной строки: `jq` (https://stedolan.github.io/jq/).

```
curl -s http://localhost:8080/api/todo | jq
[
  {
    "id": "d8d37c51-10a8-4c82-a7b1-b72b5301cdab",
    "description": "Read the Pro Spring Boot 2nd Edition Book",
```

```
    "created": "2018-05-02T22:27:26.042+0000",
    "modified": "2018-05-02T22:27:26.042+0000",
    "completed": false
  },
  {
    "id": "fbb20090-19f5-4abc-a8a9-92718c2c4759",
    "description": "Bring Milk after work",
    "created": "2018-05-02T22:28:23.249+0000",
    "modified": "2018-05-02T22:28:23.249+0000",
    "completed": false
  },
  {
    "id": "2d051b67-7716-4ee6-9c45-1de939fa579f",
    "description": "Take the dog for a walk",
    "created": "2018-05-02T22:29:28.319+0000",
    "modified": "2018-05-02T22:29:28.319+0000",
    "completed": false
  }
]
```

Теперь можно, например, модифицировать один из объектов `ToDo`:

```
curl -i -X PUT -H "Content-Type: application/json" -d '{"description":"Take
the dog and the cat for a walk",
"id":"2d051b67-7716-4ee6-9c45-1de939fa579f"}'
http://localhost:8080/api/todo
HTTP/1.1 201
Location: http://localhost:8080/api/todo/2d051b67-7716-4ee6-9c45-1de939fa579f
Content-Length: 0
Date: Wed, 02 May 2018 22:38:03 GMT
```

Фраза `Take the dog for a walk` заменена здесь на `Take the dog and the cat for a walk`. В команде используется `-X PUT`, причем необходимо указать поле `id` (получить его здесь и далее можно из заголовка `Location` из предыдущих операций `POST` или обратившись к конечной точке `/api/todo`). Если снова вывести список всех объектов `ToDo`, можно увидеть и модифицированный объект.

Пометим запланированное дело как завершенное. Выполните следующую команду:

```
curl -i -X PATCH
  http://localhost:8080/api/todo/2d051b67-7716-4ee6-9c45-1de939fa579f
HTTP/1.1 200
Location: http://localhost:8080/api/todo/2d051b67-7716-4ee6-9c45-1de939fa579f
Content-Length: 0
Date: Wed, 02 May 2018 22:50:27 GMT
```

В этой команде используется опция `-X PATCH`, которая обрабатывается методом `setCompleted`. Если вы просмотрите конечную точку, то увидите, что запланированное дело завершено.

```
curl -s http://localhost:8080/api/todo/2d051b67-7716-4ee6-9c45-1de939fa579f | jq
{
  "id": "2d051b67-7716-4ee6-9c45-1de939fa579f",
  "description": "Take the dog and the cat for a walk",
  "created": "2018-05-02T22:44:57.652+0000",
  "modified": "2018-05-02T22:50:27.691+0000",
  "completed": true
}
```

Значение поля `completed` теперь равно `true`. А раз запланированное дело завершено, его можно удалить.

```
curl -i -X DELETE http://localhost:8080/api/todo/2d051b67-7716-4ee6-9c45-
1de939fa579f
HTTP/1.1 204
Date: Wed, 02 May 2018 22:56:18 GMT
```

Команда cURL включает опцию `-X DELETE`, обрабатываемую методом `deleteToDo`, который удаляет объект `ToDo` из хеш-карты. Если теперь посмотреть на список всех запланированных дел, окажется, что их на одно меньше.

```
curl -s http://localhost:8080/api/todo | jq
[
  {
    "id": "d8d37c51-10a8-4c82-a7b1-b72b5301cdab",
    "description": "Read the Pro Spring Boot 2nd Edition Book",
    "created": "2018-05-02T22:27:26.042+0000",
    "modified": "2018-05-02T22:27:26.042+0000",
    "completed": false
  },
  {
    "id": "fbb20090-19f5-4abc-a8a9-92718c2c4759",
    "description": "Bring Milk after work",
    "created": "2018-05-02T22:28:23.249+0000",
    "modified": "2018-05-02T22:28:23.249+0000",
    "completed": false
  }
]
```

Протестируем теперь проверку корректности данных. Выполните следующую команду:

```
curl -i -X POST -H "Content-Type: application/json" -d '{"description":""}'
http://localhost:8080/api/todo
```

Эта команда отправляет данные (опция -d) с пустым полем описания (такое часто случается при отправке HTML-форм). Вы должны увидеть следующий вывод:

```
HTTP/1.1 400
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 03 May 2018 00:01:53 GMT
Connection: close
{"errors":["must not be blank"],"errorMessage":"Validation failed. 1
error(s)"}
```

Мы получили код состояния 400 («Неверный запрос»), а также сформированный классом `ToDoValidationErrorBuilder` ответ в виде `errors` и `errorMessage`. Выполните следующую команду:

```
curl -i -X POST -H "Content-Type: application/json" http://localhost:8080/
api/todo

HTTP/1.1 400
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Thu, 03 May 2018 00:07:28 GMT
Connection: close

{"errorMessage":"Required request body is missing: public org.
 springframework.http.ResponseEntity<?> com.apress.todo.controller.
 ToDoController.createToDo(com.apress.todo.domain.ToDo,org.springframework.
 validation.Errors)"}
```

Эта команда выполняет `POST` без данных, поэтому в ответ получает сообщение об ошибке, исходящее от аннотации `@ExceptionHandler` и метода `handleException`. Все ошибки (за исключением пустоты поля `description`) обрабатываются этим методом.

Можете продолжить тестирование, добавив новые запланированные дела или модифицируя аннотации проверки корректности данных, чтобы посмотреть, как они работают.

ПРИМЕЧАНИЕ

Если у вас в системе нет команды cURL или вы не можете ее установить, то воспользуйтесь любым другим REST-клиентом, например PostMan (https://www.getpostman.com) или Insomnia (https://insomnia.rest). Если же вам больше нравится командная строка, хорошим вариантом будет Httpie (https://httpie.org), использующий Python.

Spring Boot Web: переопределение настроек по умолчанию

Автоконфигурация Spring Boot Web задает настройки по умолчанию для запуска веб-приложений Spring. В этом разделе я расскажу вам, как переопределить некоторые из них.

Для переопределения веб-настроек по умолчанию можно или создать свою собственную конфигурацию (XML или JavaConfig), и/или воспользоваться файлом `application.properties` (или `.yml`).

Переопределение настроек сервера

По умолчанию встроенный Tomcat запускается на порте 8080, но это поведение можно легко поменять с помощью следующего свойства:

```
server.port=8081
```

Одна из замечательных возможностей Spring — возможность использовать SpEL (Spring Expression Language, язык выражений Spring) и применить его к этим свойствам. Например, в случае создания исполняемого JAR-файла (с помощью команды `./mvnw package` или `./gradlew build`) можно передавать параметры при запуске расширения. Пусть вы делаете следующее:

```
java -jar todo-in-memory-0.0.1-SNAPSHOT.jar --port=8787
```

и пусть в файле `application.properties` указано:

```
server.port=${port:8282}
```

Это выражение означает, что если вы передадите аргумент `--port`, то Spring им воспользуется; если же нет — параметр будет равен `8282`. Это лишь малая толика возможностей SpEL, если вы хотите узнать больше — загляните в https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#expressions.

Можно также менять адрес сервера, что удобно при необходимости запуска приложения с конкретным IP:

```
server.address=10.0.0.7
```

Можно также менять контекст приложения:

```
server.servlet.context-path=/my-todo-app
```

И можно выполнять следующие команды cURL:

```
curl -I http://localhost:8080/my-todo-app/api/todo
```

Можно также использовать SSL в Tomcat с помощью следующих свойств:

```
server.port=8443
server.ssl.key-store=classpath:keystore.jks
server.ssl.key-store-password=secret
server.ssl.key-password=secret
```

Мы вернемся к этим свойствам, и наше приложение будет работать с SSL в следующих главах.

Управлять сеансом можно с помощью следующих свойств:

```
server.servlet.session.store-dir=/tmp
server.servlet.session.persistent=true
server.servlet.session.timeout=15

server.servlet.session.cookie.name=todo-cookie.dat
server.servlet.session.cookie.path=/tmp/cookies
```

Можете также включить поддержку HTTP/2, если его поддерживает ваша среда:

```
server.http2.enabled=true
```

Формат даты JSON

По умолчанию даты отображаются в JSON-ответе в расширенном формате; но можно изменить это поведение, указав свой собственный паттерн в следующих свойствах:

```
spring.jackson.date-format=yyyy-MM-dd HH:mm:ss
spring.jackson.time-zone=MST7MDT
```

Данные свойства форматируют даты, используя при этом указанный вами часовой пояс (если хотите узнать больше о доступных идентификаторах, можете выполнить метод `java.util.TimeZone#getAvailableIDs`). Если вы модифицируете приложение ToDo, запустите его, добавьте несколько запланированных дел, выведите список — и вы должны получить следующий ответ:

```
curl -s http://localhost:8080/api/todo | jq
[
  {
    "id": "f52d1429-432d-43c5-946d-15c7fa5f50eb",
    "description": "Get the Pro Spring Boot 2nd Edition Book",
    "created": "2018-05-03 11:40:37",
    "modified": "2018-05-03 11:40:37",
    "completed": false
  }
]
```

Если хотите узнать больше о существующих свойствах, внесите в закладки ссылку https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html.

Content-Type: JSON/XML

Spring MVC использует для согласования преобразований контента при HTTP-обмене класс `HttpMessageConverters`. Spring Boot по умолчанию задает тип контента JSON, если находит в пути к классам библиотеки Jackson. Но как быть, если вы хотите выдавать также и XML и запрашивать контент в формате JSON или XML?

Spring Boot очень упрощает эту задачу благодаря дополнительной зависимости и свойству.

Если вы используете Maven, смотрите листинг 4.10.

Листинг 4.10. Maven: файл pom.xml

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

Или, если вы используете Gradle, смотрите листинг 4.11.

Листинг 4.11. Gradle: файл build.gradle

```
compile('com.fasterxml.jackson.dataformat:jackson-dataformat-xml')
```

Можете добавить следующее свойство в файл `application.properties`:

```
spring.mvc.contentnegotiation.favor-parameter=true
```

Если запустить приложение ToDo с этими изменениями, можно получить ответ в формате XML, выполнив следующую команду:

```
curl -s http://localhost:8080/api/todo?format=xml
<ArrayList><item><id>b3281340-b1aa-4104-b3d2-77a96a0e41b8</
id><description>Read the Pro Spring Boot 2nd Edition Book</
description><created>2018-05-03T19:18:30.260+0000</created><modified>2018-
05-03T19:18:30.260+0000</modified><completed>false</completed></item></
ArrayList>
```

В предыдущей команде к URL было добавлено `?format=xml`; аналогично можно получить ответ в формате JSON:

```
curl -s http://localhost:8080/api/todo?format=json | jq
[
  {
    "id": "b3281340-b1aa-4104-b3d2-77a96a0e41b8",
    "description": "Read the Pro Spring Boot 2nd Edition Book",
    "created": "2018-05-03T19:18:30.260+0000",
    "modified": "2018-05-03T19:18:30.260+0000",
    "completed": false
  }
]
```

Если хотите красиво отформатировать выводимый XML, можете воспользоваться существующей в Unix-средах командой `xmllint`[1]:

```
curl -s http://localhost:8080/api/todo?format=xml | xmllint --format -
```

Spring MVC: переопределение настроек по умолчанию

До сих пор я не демонстрировал вам создание приложений, сочетающих в клиентской части несколько технологий вроде HTML, JavaScript и т. д., поскольку сегодня компании — разработчики ПО склоняются к использованию для клиентской части JavaScript/TypeScript.

Это не значит, что нельзя создать приложение Spring Web MVC с прикладной и клиентской частями. Spring Web MVC отлично интегрируется с шаблонизаторами и другими технологиями.

[1] В Windows ее тоже можно установить (https://www.zlatkovic.com/libxml.en.html). Учтите, что, кроме самой xmllint, вам понадобятся также библиотеки icov и zlib.

При применении любого шаблонизатора можно выбрать префикс и суффикс для представления с помощью следующих свойств:

```
spring.mvc.view.prefix=/WEB-INF/my-views/
spring.mvc.view.suffix=.jsp
```

Как вы видите, Spring Boot может помочь со множеством конфигурационных настроек, которых при создании обычных приложений Spring Web необходимо очень много. Благодаря этому новому подходу разработка приложений ускоряется. Больше инструкций по применению Spring MVC и всех его возможностей можно найти в справочной документации: https://docs.spring.io/spring/docs/5.0.5.RELEASE/spring-framework-reference/web.html#mvc.

Использование другого контейнера приложения

По умолчанию Spring Boot использует в качестве контейнера приложений Tomcat (для веб-приложений-сервлетов) и настраивает встроенный сервер. Для переопределения этого поведения можно модифицировать файл `pom.xml` Maven или файл `build.gradle` Gradle.

Использование Jetty Server

Для Undertow и Netty нужны примерно одинаковые изменения (листинги 4.12 и 4.13).

Листинг 4.12. Maven — файл pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
      <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
            </exclusion>
      </exclusions>
</dependency>
<dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Листинг 4.13. Gradle: файл build.gradle

```
configurations {
 compile.exclude module: "spring-boot-starter-tomcat"
}

dependencies {
 compile("org.springframework.boot:spring-boot-starter-web")
 compile("org.springframework.boot:spring-boot-starter-jetty")
  // ...
}
```

Spring Boot Web: клиент

Еще одна важная возможность создания веб-приложений с помощью Spring Boot — наличие в Spring Web MVC удобного класса `RestTemplate` для создания клиентов.

Клиентское приложение ToDo

Откройте браузер и перейдите по адресу https://start.spring.io для создания клиентского приложения ToDo на основе следующих значений (рис. 4.2).

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-client`.
- Name (Название): `todo-client`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Web`, `Lombok`.

Нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в вашу интегрированную среду разработки.

Модель предметной области: ToDo

Создайте модель предметной области ToDo, соответствующую минимальному набору полей из предыдущего приложения (листинг 4.14).

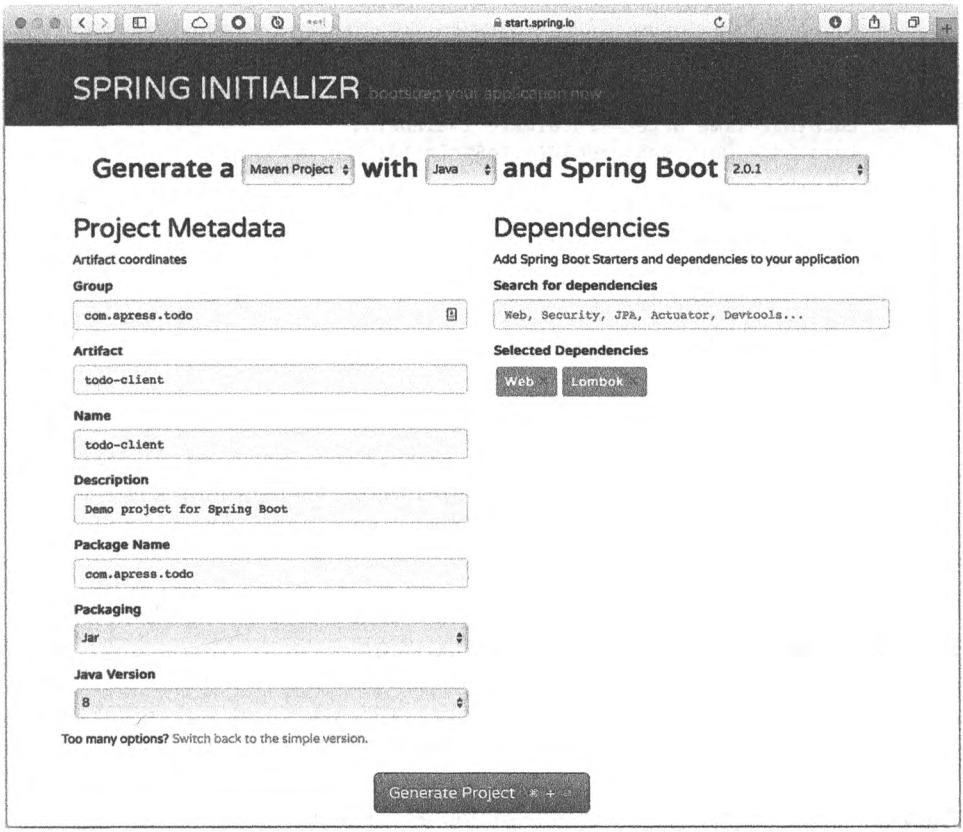


Рис. 4.2. Клиентское приложение ToDo

Листинг 4.14. com.apress.todo.client.domain.ToDo.java

```
package com.apress.todo.client.domain;

import lombok.Data;

import java.time.LocalDateTime;
import java.util.UUID;

@Data
public class ToDo {

    private String id;
    private String description;
    private LocalDateTime created;
```

```
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
        LocalDateTime date = LocalDateTime.now();
        this.id = UUID.randomUUID().toString();
        this.created = date;
        this.modified = date;
    }

    public ToDo(String description){
        this();
        this.description = description;
    }
}
```

В листинге 4.14 приведен класс модели предметной области ToDo. Название пакета — другое, оно не обязано соответствовать названию класса. Приложению не обязательно знать, какой пакет сериализовать/десериализовать в/из формата JSON.

Обработчик ошибок: ToDoErrorHandler

Создадим обработчик ошибок, который позаботится обо всех поступающих от сервера сообщениях об ошибках. Создайте класс `ToDoErrorHandler` (листинг 4.15).

Листинг 4.15. com.apress.todo.client.error.ToDoErrorHandler.java

```
package com.apress.todo.client.error;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import org.springframework.http.client.ClientHttpResponse;
import org.springframework.util.StreamUtils;
import org.springframework.web.client.DefaultResponseErrorHandler;
import java.io.IOException;
import java.nio.charset.Charset;

public class ToDoErrorHandler extends DefaultResponseErrorHandler {

    private Logger log = LoggerFactory.getLogger(ToDoErrorHandler.class);

    @Override
    public void handleError(ClientHttpResponse response)
      throws IOException {
        log.error(response.getStatusCode().toString());
```

```
        log.error(StreamUtils.copyToString(response.getBody(),
                                           Charset.defaultCharset()));
    }
}
```

В листинге 4.15 показан класс `ToDoErrorHandler` — пользовательский класс, расширяющий стандартный класс `DefaultResponseErrorHandler`. При получении HTTP-состояния 400 («Неверный запрос») можно перехватить ошибку и отреагировать на нее должным образом; но в данном случае класс просто заносит сообщение об ошибке в журнал.

Пользовательские свойства: класс ToDoRestClientProperties

Важно знать, где именно запущено приложение ToDo и какой basePath использовать. Поэтому необходимо где-то хранить данную информацию. Рекомендуемая практика: внешнее хранение этой информации.

Создадим класс `ToDoRestClientProperties` для информации об URL и basePath. Эту информацию можно сохранить в файле `application.properties` (листинг 4.16).

Листинг 4.16. com.apress.todo.client.ToDoRestClientProperties.java

```
package com.apress.todo.client;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;

@Component
@ConfigurationProperties(prefix="todo")
@Data
public class ToDoRestClientProperties {

    private String url;
    private String basePath;
}
```

В листинге 4.16 приведен класс `ToDoRestClientProperties`, содержащий информацию об URL и basePath. Spring Boot позволяет создавать настраиваемые типизированные свойства, к которым можно получить доступ из приложения и которые можно связать с файлом `application.properties`; единственное требование — этот класс необходимо снабдить аннотацией `@ConfigurationProperties`, которая может принимать такие параметры, как префикс.

Добавьте в файл `application.properties` следующее содержимое:

```
todo.url=http://localhost:8080
todo.base-path=/api/todo
```

Клиент: класс ToDoRestClient

Создадим клиент, который бы использовал класс `RestTemplate`, упрощающий обмен информацией между этим клиентом и сервером. Создайте класс `ToDoRestClient` (листинг 4.17).

Листинг 4.17. com.apress.todo.client.ToDoRestClient.java

```
package com.apress.todo.client;

import com.apress.todo.client.domain.ToDo;
import com.apress.todo.client.error.ToDoErrorHandler;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.http.*;
import org.springframework.stereotype.Service;
import org.springframework.web.client.RestTemplate;

import java.net.URI;
import java.net.URISyntaxException;
import java.util.HashMap;
import java.util.Map;

@Service
public class ToDoRestClient {

    private RestTemplate restTemplate;
    private ToDoRestClientProperties properties;

    public ToDoRestClient(
                        ToDoRestClientProperties properties){
        this.restTemplate = new RestTemplate();
        this.restTemplate.setErrorHandler(
                                        new ToDoErrorHandler());
        this.properties = properties;
    }

    public Iterable<ToDo> findAll() throws URISyntaxException {
        RequestEntity<Iterable<ToDo>> requestEntity = new RequestEntity
        <Iterable<ToDo>>(HttpMethod.GET,new URI(properties.getUrl() +
        properties.getBasePath()));
        ResponseEntity<Iterable<ToDo>> response =
                restTemplate.exchange(requestEntity,new
                    ParameterizedTypeReference<Iterable<ToDo>>(){});
```

```
        if(response.getStatusCode() == HttpStatus.OK){
            return response.getBody();
        }

        return null;
    }
    public ToDo findById(String id){
        Map<String, String> params = new HashMap<String, String>();
        params.put("id", id);
        return restTemplate.getForObject(properties.getUrl() +
               properties.getBasePath() + "/{id}",ToDo.class,params);
    }

    public ToDo upsert(ToDo toDo) throws URISyntaxException {
        RequestEntity<?> requestEntity = new RequestEntity<>(toDo, HttpMethod.
            POST, new URI(properties.getUrl() + properties.getBasePath()));
        ResponseEntity<?> response = restTemplate.exchange(requestEntity, new
            ParameterizedTypeReference<ToDo>() {});

       if(response.getStatusCode() == HttpStatus.CREATED){
            return restTemplate.getForObject(response.getHeaders().
                getLocation(),ToDo.class);
       }
       return null;
    }

    public ToDo setCompleted(String id) throws URISyntaxException{
        Map<String, String> params = new HashMap<String, String>();
        params.put("id", id);
        restTemplate.postForObject(properties.getUrl() + properties.getBase
            Path() + "/{id}?_method=patch",null,ResponseEntity.class, params);
        return findById(id);
    }

    public void delete(String id){
        Map<String, String> params = new HashMap<String, String>();
        params.put("id", id);
        restTemplate.delete(properties.getUrl() + properties.getBasePath() +
            "/{id}",params);
    }
}
```

В листинге 4.17 показан взаимодействующий с приложением ToDo клиент. Этот класс использует класс `RestTemplate`. **`RestTemplate`** — основной класс Spring для синхронного HTTP-доступа на стороне клиента. Он упрощает взаимодействие между HTTP-серверами и обеспечивает соблюдение принципов REST. Он также отвечает за установление HTTP-соединений, благодаря чему от кода приложения требуется только предоставлять URL (возможно, с переменными шаблона) и извлекать результаты. Еще одна из множества

его возможностей — описание своих пользовательских реакций на ошибки. Взгляните на его конструктор, и вы увидите, что в качестве обработчика ошибок задан объект `ToDoErrorHandler`.

Посмотрите на класс внимательнее; он включает все те же действия, что и приложение ToDo (прикладная часть).

ПРИМЕЧАНИЕ

По умолчанию для установления HTTP-соединений класс RestTemplate задействует стандартные функции JDK. Можно перейти на использование и других HTTP-библиотек, например Apache HttpComponents, Netty и OkHttp, с помощью метода InterceptingHttpAccessor.setRequestFactory(org.springframework.http.client.ClienthttprequestFactory).

Запуск и тестирование клиента

Для запуска и тестирования клиента модифицируйте[1] класс `TodoClientApplication` так, как показано в листинге 4.18.

Листинг 4.18. com.apress.todo.TodoClientApplication.java

```
package com.apress.todo;

import com.apress.todo.client.ToDoRestClient;
import com.apress.todo.client.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.WebApplicationType;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@SpringBootApplication
public class TodoClientApplication {

      public static void main(String[] args) {
            SpringApplication app = new
                SpringApplication(TodoClientApplication.class);
            app.setWebApplicationType(WebApplicationType.NONE);
            app.run(args);
      }
```

[1] Видимо, автор имел в виду «создайте».

```
    private Logger log = LoggerFactory.getLogger
        (TodoClientApplication.class);

    @Bean
    public CommandLineRunner process(ToDoRestClient client){
        return args -> {

          Iterable<ToDo> toDos = client.findAll();
          assert toDos != null;
          toDos.forEach( toDo -> log.info(toDo.toString()));

          ToDo newToDo = client.upsert(new ToDo("Drink plenty of Water
              daily!"));
          assert newToDo != null;
          log.info(newToDo.toString());

          ToDo toDo = client.findById(newToDo.getId());
          assert toDos != null;
          log.info(toDo.toString());

          ToDo completed = client.setCompleted(newToDo.getId());
          assert completed.isCompleted();
          log.info(completed.toString());

          client.delete(newToDo.getId());
          assert client.findById(newToDo.getId()) == null;
       };
   }
}
```

В листинге 4.18 показано, как протестировать клиент. Во-первых, отключаем веб-среду с помощью `WebApplicationType.NONE`. Затем с помощью `CommandLineRunner` (в качестве компонента) выполняем код перед запуском приложения.

Прежде чем запустить этот код, проанализируйте его, чтобы понимать, что в нем происходит. Запустить его можно с помощью командной строки или IDE. Убедитесь, что приложение ToDo запущено и работает.

Мы еще воспользуемся этим клиентом.

ПРИМЕЧАНИЕ

Напоминаю, что весь исходный код для данной книги можно скачать с веб-сайта Apress или из GitHub: https://github.com/Apress/pro-spring-boot-2.

Резюме

Из этой главы вы узнали, как Spring Boot управляет веб-приложениями, производит их автоматическую конфигурацию и использует все возможности Spring MVC. Вы также узнали, как можно переопределить все создаваемые автоконфигурацией по умолчанию продуманные настройки.

На примере приложения ToDo я показал часть возможностей Spring Boot по созданию веб-приложений, в частности JSON- и XML-конфигурации, использование нескольких аннотаций MVC, например `@RequestMapping` и `@ExceptionHandler`, и многое другое.

Вы также узнали, как Spring Boot задействует встроенный контейнер приложений (в данном случае используемый по умолчанию Tomcat) для простоты развертывания и повышения переносимости.

Все указанные в приложении ToDo аннотации — часть Spring MVC. В оставшейся части данной книги мы реализуем и другие. А если вы хотите узнать больше о Spring MVC, загляните в книгу *Pro Spring 5* («Spring 5 для профессионалов»).

Из следующей главы вы узнаете, как Spring Boot производит автоконфигурацию модулей Spring Data для систем постоянного хранения. Я также расскажу о модулях Spring JDBC, JPA, REST и NoSQL (таких как Mongo).

5 Доступ к данным

Данные стали важнейшей частью вселенной IT, начиная с попыток доступа к ним, их сохранения и анализа и до использования от нескольких байтов до петабайтов информации. Было предпринято немало попыток создания фреймворков и библиотек для упрощения работы разработчиков с данными, но иногда эта задача все равно оказывается слишком сложной.

После версии 3.0 фреймворк Spring сформировал несколько команд, которые специализировались на различных технологиях. Так появилась команда Spring Data. Цель этого конкретного проекта — упростить использование технологий доступа к данным, начиная с реляционных и нереляционных баз данных и до фреймворков отображения-свертки (map-reduce) и облачных сервисов обработки и передачи данных. Проект Spring Data представляет собой набор подпроектов, ориентированных на работу с конкретными базами данных.

В этой главе описывается доступ к данным с помощью Spring Boot на примере приложения ToDo из предыдущих глав. Скоро мы научим приложение ToDo работать с SQL- и NoSQL-базами данных. Что ж, приступим.

Базы данных SQL

Помните время, когда (в мире Java) приходилось вручную решать все задачи JDBC (Java Database Connectivity) — соединения с базами данных на Java? Тогда приходилось скачивать нужные драйверы и указывать строки подключения, открывать и закрывать соединения, SQL-операторы, результирующие наборы данных и транзакции и преобразовывать результирующие наборы

данных в объекты. Мне кажется, что все это слишком ручная работа. Позднее начали появляться многочисленные фреймворки объектно-реляционного отображения (ORM), предназначенные для автоматизации этих задач — такие фреймворки, например, как Castor XML, ObjectStore и Hibernate. С их помощью можно было распознавать классы предметной области и создавать соответствующий таблицам базы данных код XML. На каком-то этапе оказалось, что для работы с подобными фреймворками также нужно быть экспертом.

Фреймворк Spring очень сильно помог в работе с такими фреймворками за счет следования *паттерну проектирования «Шаблонный метод»* (template design pattern). В его основе лежит создание абстрактного класса, определяющего способы выполнения методов и создающего абстракции баз данных, благодаря чему разработчик может сосредоточить свое внимание только на бизнес-логике. Наиболее трудную часть работы при этом берет на себя фреймворк Spring, включая выполнение соединений (открытие, закрытие и организацию их пула), транзакций и взаимодействие с фреймворками.

Имеет смысл упомянуть, что фреймворк Spring опирается на несколько интерфейсов и классов (например, интерфейс `javax.sql.DataSource`) для получения информации об используемой базе данных, способе подключения к ней (строке подключения) и учетных данных. Если требуется управление транзакциями, то без интерфейса `DataSource` не обойтись. Обычно для интерфейса `DataSource` необходим класс `Driver`, URL JDBC, а также имя пользователя и пароль для подключения к базе данных.

Spring Data

Команда разработчиков Spring Data создала несколько замечательных, ориентированных на работу с данными фреймворков, доступных сегодня для использования сообществом Java и Spring. Они ставили перед собой задачу обеспечить возможность в привычном и единообразном стиле программировать на Spring операции доступа к данным и полностью контролировать используемую «за кулисами» технологию их хранения.

В проекте Spring Data сгруппированы несколько дополнительных библиотек и фреймворков по работе с данными, что упрощает использование технологий доступа к данным для реляционных и нереляционных (NoSQL) баз данных.

Вот некоторые из возможностей Spring Data.

- Поддержка хранения данных с использованием нескольких хранилищ.
- Абстракции объектного отображения: из репозитория и пользовательские.
- Динамическое выполнение запросов на основе названий методов.
- Удобная интеграция со Spring через JavaConfig и XML.
- Поддержка контроллеров Spring MVC.
- Наличие событий, обеспечивающих возможность прозрачного аудита (создание, недавние изменения).

Это лишь малая часть возможностей — для описания их всех понадобилась бы отдельная книга. В этой главе мы обсудим ровно столько, сколько нужно для создания полнофункциональных, ориентированных на работу с данными приложений. Помните, что все описанные проекты объединены под общим проектом Spring Data.

Spring JDBC

В этом разделе я покажу вам, как пользоваться классом `JdbcTemplate`. Данный класс реализует паттерн проектирования «Шаблонный метод» и представляет собой конкретный класс, предоставляющий несколько заданных способов (шаблонов) выполнения его методов. Весь стереотипный код алгоритмов (наборы инструкций) скрыт. В Spring можно выбирать различные способы формирования фундамента доступа к базе данных через JDBC; на самом низком уровне располагается классический подход JDBC Spring — класс `JdbcTemplate`.

При использовании класса `JdbcTemplate` для воплощения простого способа взаимодействия с любыми СУБД достаточно реализовать интерфейсы обратного вызова. Для класса `JdbcTemplate` требуется интерфейс `javax.sql.DataSource`, его можно использовать в любом классе, просто объявив его в JavaConfig, XML или с помощью аннотаций. Класс `JdbcTemplate` берет на себя должную обработку всех исключений `SQLException`.

Для указания поименованных параметров (`:parameterName`) вместо обычных JDBC-«заполнителей» `"?"` можно использовать класс `NamedParameterJdbcTemplate` (JDBC-адаптер). Эта еще одна возможность, которую можно применять в своих SQL-запросах.

Класс `JdbcTemplate` предоставляет для использования различные методы.

- Для выполнения запросов (`SELECT`) обычно задействуются методы `query`, `queryForObject`.
- Для обновления (`INSERT/UPDATE/DELETE`) используется метод `update`.
- Для различных операций (над базой данных/таблицей/для функций) применяются методы `execute` и `update`.

Благодаря Spring JDBC можно вызывать хранимые процедуры с помощью класса `SimpleJdbcCall` и производить над результатом различные операции с помощью конкретного интерфейса `RowMapper`. Класс `JdbcTemplate` использует `RowMapper<T>` для построчного отображения (задания соответствий) строк `ResultSet`.

Spring JDBC поддерживает встроенные СУБД, например HSQL, H2 и Derby. Настройка очень проста, а запуск и тестирование не занимают много времени.

Еще одна возможность — инициализация баз данных с помощью скриптов; причем можно или использовать встроенную поддержку, или не использовать. Можно добавлять свои собственные схемы и данные в SQL-формате.

Работа с JDBC в Spring Boot

Фреймворк Spring поддерживает работу с базами данных с помощью JDBC или ORM (объектно-реляционных отображений). Spring Boot еще более расширяет возможности работающих с данными приложений.

Spring Boot использует автоконфигурацию для задания разумных настроек по умолчанию, если обнаруживает, что в приложении есть JAR-файлы JDBC. Spring Boot автоматически конфигурирует источник данных в соответствии с найденным по пути к классам SQL-драйвером. Если он обнаруживает наличие любой встроенной СУБД (H2, HSQL или Derby), то использует для нее настройки по умолчанию; другими словами, в приложении может быть две зависимости для драйверов (например, MySQL и H2) и если Spring Boot не найдет никакого объявления компонента для источника данных, то создаст его на основе найденного по пути к классам JAR-файла встроенной СУБД (например, H2). Spring Boot также настраивает в качестве пула соединений по умолчанию HikariCP. Конечно, эти настройки по умолчанию можно переопределить.

Для переопределения этих настроек по умолчанию необходимо создать свое собственное объявление источника данных в виде JavaConfig, XML или в файле `application.properties`.

```
# Пользовательский DataSource
spring.datasource.username=springboot
spring.datasource.password=rocks!
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/testdb?autoReconnect=
    true&useSSL=false
```

При развертывании приложения в контейнере приложений Spring Boot поддерживает соединения JNDI. Задать имя JNDI можно в файле `application.properties`:

```
spring.datasource.jndi-name=java:jboss/ds/todos
```

Еще одна возможность Spring Boot, удобная для работающих с данными приложений: при наличии по пути к классам файлов `schema.sql`, `data.sql`, `schema-<платформа>.sql` или `data-<платформа>.sql` Spring Boot инициализирует базу данных на их основе.

Итак, для использования JDBC в приложении Spring Boot необходимо добавить зависимость `spring-boot-starter-jdbc` и SQL-драйвер.

Приложение ToDo с использованием JDBC

Пришло время поработать с приложением ToDo, как мы делали в предыдущей главе. Можете начать с нуля или воспользоваться прилагаемым кодом[1]. Если начинаете с нуля, то перейдите в Spring Initializr (https://start.spring.io) и внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-jdbc`.
- Name (Название): `todo-jdbc`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Web, Lombok, JDBC, H2, MySQL`.

[1] Этот и несколько следующих примеров из этой главы не работают на Java 11 без внесения дополнительных изменений (из-за проблем совместимости версий Lombok и некоторых других пакетов, например jaxb-api). На Java 8 все работает.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами интегрированную среду разработки (рис. 5.1).

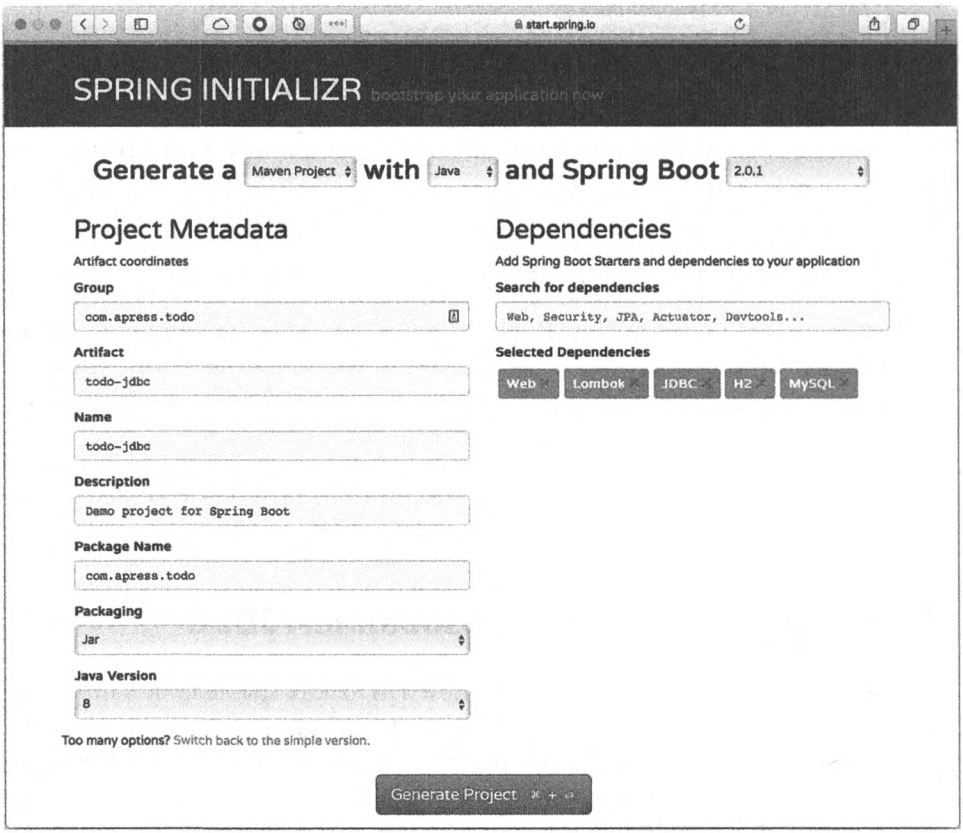


Рис. 5.1. Spring Initializr (https://start.spring.io)

Можете скопировать все классы из предыдущей главы, за исключением класса `ToDoRepository` — это единственный модифицированный класс. Убедитесь также, что в файле `pom.xml` или `build.gradle` есть два драйвера: H2 и MySQL. Что сделает автоконфигурация Spring Boot, согласно упомянутому в предыдущем разделе, если не указать никакого источника данных (в JavaConfig, XML или в файле `application.properties`)? Правильно! Spring Boot автоматически настроит по умолчанию встроенную базу данных H2.

Репозиторий: класс ToDoRepository

Создайте класс `ToDoRepository`, реализующий интерфейс `CommonRepository` (листинг 5.1).

Листинг 5.1. com.apress.todo.respository.ToDoRepository.java

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.dao.EmptyResultDataAccessException;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
import org.springframework.stereotype.Repository;

import java.sql.ResultSet;
import java.time.LocalDateTime;
import java.util.*;

@Repository
public class ToDoRepository implements CommonRepository<ToDo> {

     private static final String SQL_INSERT = "insert into todo (id,
         description, created, modified, completed) values (:id,:description,
         :created,:modified,:completed)";
     private static final String SQL_QUERY_FIND_ALL = "select id, description,
         created, modified, completed from todo";
     private static final String SQL_QUERY_FIND_BY_ID = SQL_QUERY_FIND_ALL + "
         where id = :id";
     private static final String SQL_UPDATE = "update todo set description =
         :description, modified = :modified, completed = :completed
         where id = :id";
     private static final String SQL_DELETE = "delete from todo where id = :id";

    private final NamedParameterJdbcTemplate jdbcTemplate;

    public ToDoRepository(NamedParameterJdbcTemplate jdbcTemplate){
        this.jdbcTemplate = jdbcTemplate;
    }

    private RowMapper<ToDo> toDoRowMapper = (ResultSet rs, int rowNum) -> {
        ToDo toDo = new ToDo();
        toDo.setId(rs.getString("id"));
        toDo.setDescription(rs.getString("description"));
        toDo.setModified(rs.getTimestamp("modified").toLocalDateTime());
        toDo.setCreated(rs.getTimestamp("created").toLocalDateTime());
        toDo.setCompleted(rs.getBoolean("completed"));
        return toDo;
    };
```

```
@Override
public ToDo save(final ToDo domain) {
    ToDo result = findById(domain.getId());
    if(result != null){
        result.setDescription(domain.getDescription());
        result.setCompleted(domain.isCompleted());
        result.setModified(LocalDateTime.now());
        return upsert(result, SQL_UPDATE);
    }
    return upsert(domain,SQL_INSERT);
}

private ToDo upsert(final ToDo toDo, final String sql){
    Map<String, Object> namedParameters = new HashMap<>();
    namedParameters.put("id",toDo.getId());
    namedParameters.put("description",toDo.getDescription());
    namedParameters.put("created",java.sql.Timestamp.valueOf(toDo.
        getCreated()));
    namedParameters.put("modified",java.sql.Timestamp.valueOf(toDo.
        getModified()));
    namedParameters.put("completed",toDo.isCompleted());
    this.jdbcTemplate.update(sql,namedParameters);
    return findById(toDo.getId());
}

@Override
public Iterable<ToDo> save(Collection<ToDo> domains) {
    domains.forEach( this::save);
    return findAll();
}

@Override
public void delete(final ToDo domain) {
     Map<String, String> namedParameters = Collections.singletonMap("id",
         domain.getId());
    this.jdbcTemplate.update(SQL_DELETE,namedParameters);
}

@Override
public ToDo findById(String id) {
    try {
         Map<String, String> namedParameters = Collections.
             singletonMap("id", id);
         return this.jdbcTemplate.queryForObject(SQL_QUERY_FIND_BY_ID,
             namedParameters, toDoRowMapper);
    } catch (EmptyResultDataAccessException ex) {
        return null;
    }
}
```

```
    @Override
    public Iterable<ToDo> findAll() {
        return this.jdbcTemplate.query(SQL_QUERY_FIND_ALL, toDoRowMapper);
    }
}
```

В листинге 5.1 приведен класс `ToDoRepository`, использующий `JdbcTemplate`, хотя и не напрямую. Этот класс задействует класс `NamedParameterJdbcTemplate` (JDBC-адаптер), с помощью которого можно работать с поименованными параметрами, то есть вместо `?` в SQL-запросе можно использовать названия вроде `:id`.

В этом классе также объявлен объект `RowMapper`; напомню, что `JdbcTemplate` применяет `RowMapper<T>` для построчного отображения (задания соответствий) строк `ResultSet`.

Проанализируйте приведенный код и отметьте для себя все реализации методов, использующие чистый SQL.

Инициализация базы данных: schema.sql

Как вы помните, фреймворк Spring позволяет инициализировать базу данных — создавать таблицы или менять их схемы, вставлять/обновлять данные при запуске приложения. Для инициализации приложения Spring (не Spring Boot) необходимо добавить конфигурацию (XML или JavaConfig), но приложение ToDo — это приложение Spring Boot. Если Spring Boot находит файлы `schema.sql` и/или `data.sql`, то автоматически их выполняет. Создадим файл `schema.sql` (листинг 5.2).

Листинг 5.2. Файл schema.sql

```
DROP TABLE IF EXISTS todo;
CREATE TABLE todo
(
  id varchar(36) not null primary key,
  description varchar(255) not null,
  created timestamp,
  modified timestamp,
  completed boolean
);
```

Листинг 5.2 демонстрирует файл `schema.sql`, выполняемый при запуске приложения, а поскольку H2 — источник данных по умолчанию, то этот сценарий выполняется для базы данных H2.

Запуск и тестирование: приложение ToDo

Пришло время запустить и протестировать приложение ToDo. Можете запустить его в своей IDE или, если вы используете Maven, выполнить команду:

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

Если вы используете Gradle, выполните:

```
./gradlew bootRun
```

Для тестирования приложения ToDo можете запустить свое приложение ToDoClient. Оно должно работать без всяких проблем.

Консоль H2

Как убедиться после запуска приложения ToDo, что оно сохраняет данные в базе данных H2? В Spring Boot есть свойство для включения консоли H2, с которой вы сможете работать. Она очень удобна для разработки (но не для среды промышленной эксплуатации).

Модифицируйте файл `application.properties` и добавьте в него следующее свойство:

```
spring.h2.console.enabled=true
```

Перезапустите приложение ToDo, добавьте в него значения с помощью команды `cURL`, после чего перейдите в браузер и откройте страницу http://localhost:8080/h2-console (рис. 5.2).

На рис. 5.2 показана консоль H2, доступная на конечной точке `/h2-console` (эту конечную точку также можно переопределить). URL JDBC должен быть `jdbc:h2:mem:testdb` (иногда он отличается, так что поменяйте его на это значение). По умолчанию название базы данных `testdb` (но его тоже можно переопределить). Нажав кнопку Connect (Подключиться), можно получить другой экран, на котором будет видна таблица с созданными данными (рис. 5.3).

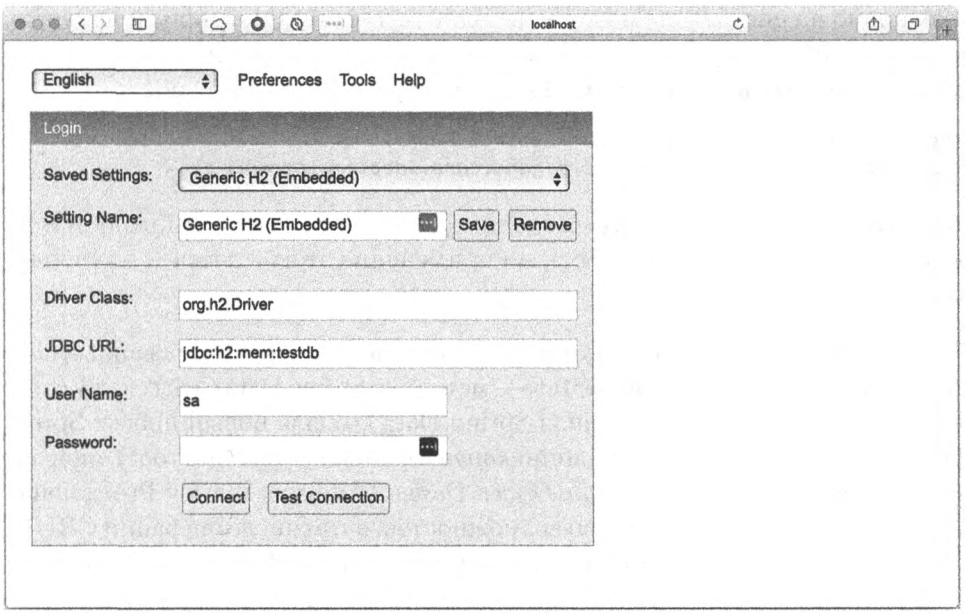


Рис. 5.2. http://localhost:8080/h2-console

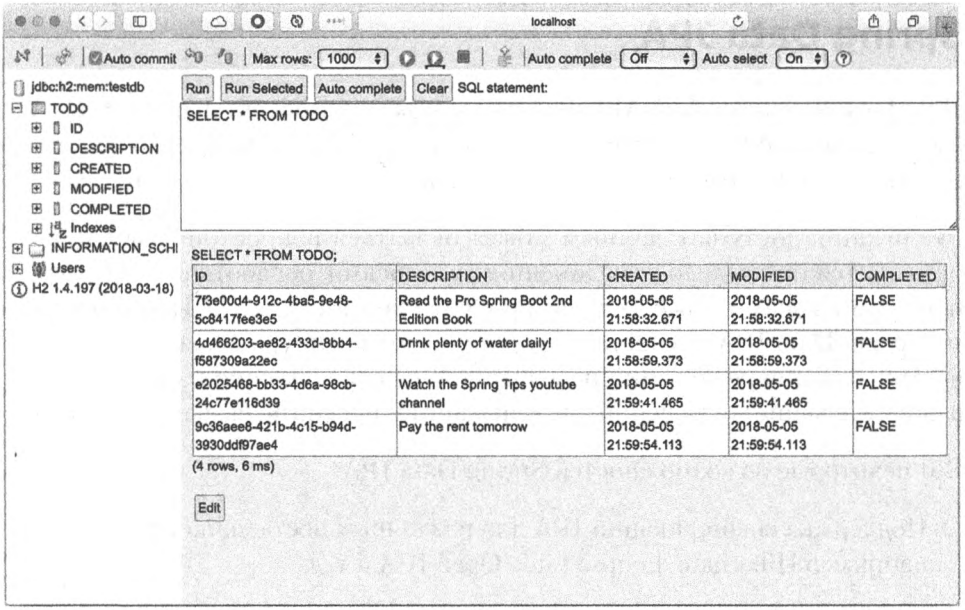


Рис. 5.3. http://localhost:8080/h2-console

Как видно на рис. 5.3, вы можете выполнить любой SQL-запрос и получить данные. Чтобы видеть, какие SQL-запросы выполняются в приложении ToDo, можно добавить в файл `application.properties` следующие свойства:

```
logging.level.org.springframework.data=INFO
logging.level.org.springframework.jdbc.core.JdbcTemplate=DEBUG
```

Как вы видите, класс `JdbcTemplate` предоставляет массу возможностей по взаимодействию с любой СУБД, но применение этого класса — «наиболее низкоуровневый» подход.

На момент написания данной книги появился новый, унифицированный способ применения класса `JdbcTemplate` — методика Spring Data (я расскажу о ней в следующих разделах). Команда Spring Data создала новый проект Spring Data JDBC, следующий концепции *корня агрегата* (aggregate root), описанной в книге Eric Evans, *Domain-Driven Design* (Addison-Wesley Professional, 2003)[1]. Он включает множество возможностей, в том числе операции CRUD, поддержку аннотаций `@Query`, поддержку запросов MyBatis, событий и много другого, так что рекомендуем следить за этим проектом. Это принципиально новый подход к JDBC.

Spring Data JPA

JPA (Java Persistence API, API постоянного хранения объектов Java) обеспечивает модель персистентности POJO для объектно-реляционных отображений. Spring Data JPA позволяет хранить данные на основе этой модели.

Реализация доступа к данным может оказаться непростой задачей, ведь приходится иметь дело с соединениями, сеансами, обработкой исключений и другими вещами, даже в случае простых операций CRUD. Именно поэтому Spring Data JPA предоставляет дополнительный уровень функциональности: создание реализаций репозитория непосредственно из интерфейсов и использование соглашений для генерации запросов по названиям методов.

Вот некоторые из возможностей Spring Data JPA.

- Поддержка спецификации JPA для различных поставщиков реализаций, например Hibernate, Eclipse Link, Open JPA и т. д.

[1] *Эванс Э.* Предметно-ориентированное проектирование (DDD): Структуризация сложных программных систем. — М.: Вильямс, 2011.

- Поддержка репозиториев (концепция из *Domain-Driven Design*).
- Аудит класса предметной области.
- Поддержка предикатов Quesydsl (http://www.querydsl.com/) и типобезопасных JPA-запросов.
- Постраничный вывод, сортировка, поддержка выполнения динамических запросов.
- Поддержка аннотаций `@Query`.
- Поддержка отображения сущностей на основе XML.
- Конфигурация репозитория на основе JavaConfig с помощью аннотации `@EnableJpaRepositories`.

Использование Spring Data JPA со Spring Boot

Одно из главных преимуществ Spring Data JPA состоит в том, что можно не заботиться о реализации основных функциональных возможностей CRUD. Необходимо только создать интерфейс, расширяющий `Repository<T,ID>`, `CrudRepository<T,ID>` или `JpaRepository<T,ID>`. Интерфейс `JpaRepository` не только предоставляет функциональность CRUD, но и расширяет интерфейс `PagingAndSortingRepository`, включающий дополнительную функциональность. Если вы взглянете на описание интерфейса `CrudRepository<T,ID>` (используемого в нашем приложении ToDo), то увидите все классические методы, как показано в листинге 5.3.

Листинг 5.3. org.springframework.data.repository.CrudRepository.java

```
@NoRepositoryBean
public interface CrudRepository<T, ID> extends Repository<T, ID> {
      <S extends T> S save(S entity);
      <S extends T> Iterable<S> saveAll(Iterable<S> entities);
      Optional<T> findById(ID id);
      boolean existsById(ID id);
      Iterable<T> findAll();
      Iterable<T> findAllById(Iterable<ID> ids);
      long count();
      void deleteById(ID id);
      void delete(T entity);
      void deleteAll(Iterable<? extends T> entities);
      void deleteAll();
}
```

В листинге 5.3 приведено описание интерфейса `CrudRepository<T,ID>`, где `T` означает сущность (класс модели предметной области), а `ID` — первичный ключ, который должен реализовать интерфейс `Serializable`.

В простых приложениях Spring необходимо использовать аннотацию `@EnableJpaRepositories`, инициирующую процедуру дополнительной конфигурации, применяемую во время жизненного цикла описанных в приложении репозиториев. Хорошая новость: при использовании Spring Boot этого не требуется, поскольку Spring Boot берет эту работу на себя. Еще одна возможность Spring Data JPA — методы запросов, обладающий огромным потенциалом способ создания SQL-операторов с полями сущностей предметной области.

Итак, для использования Spring Data JPA со Spring Boot необходимы зависимость `spring-boot-starter-data-jpa` и SQL-драйвер.

Когда Spring Boot производит автоконфигурацию и обнаруживает наличие JAR-файлов Spring Data JPA, он настраивает источник данных по умолчанию (если ни один не описан). Настраивает поставщик JPA (по умолчанию используется Hibernate), активирует репозитории (с помощью аннотации `@EnableJpaRepositories`). Проверяет, описаны ли какие-нибудь методы запросов. И делает многое другое.

Создание приложения ToDo с использованием Spring Data JPA

Можете создать приложение ToDo с нуля или просто посмотреть на нужные классы, а также необходимые зависимости в файлах `pom.xml` и `build.gradle`.

Если вы начинаете с нуля, перейдите в браузер и откройте Spring Initializr. Внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-jpa`.
- Name (Название): `todo-jpa`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Web`, `Lombok`, `JPA`, `H2`, `MySQL`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами IDE (рис. 5.4).

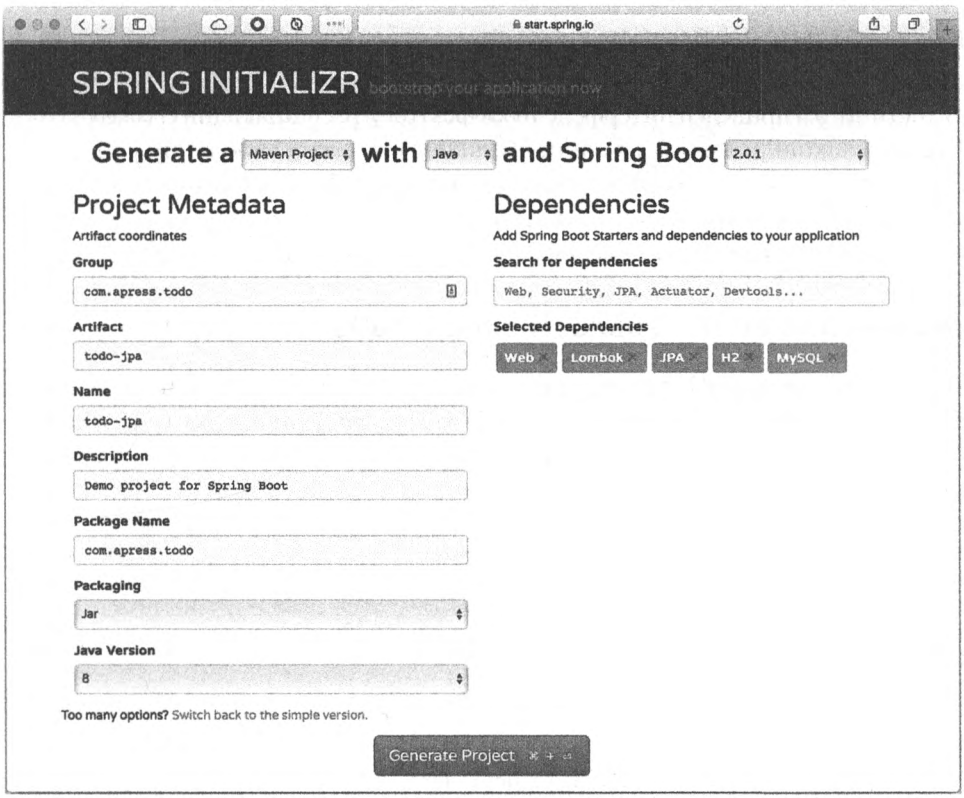


Рис. 5.4. Spring Initializr (https://start.spring.io)

Можете скопировать все классы из предыдущей главы, за исключением класса `ToDoRepository` — это единственный новый класс[1]; все остальные мы изменим.

Репозиторий: ToDoRepository

Создайте интерфейс `ToDoRepository`, расширяющий `CrudRepository<T,ID>`. Роль `T` здесь играет класс `ToDo`, а `ID` — тип `String` (листинг 5.4).

[1] Теперь класс заменен интерфейсом.

Листинг 5.4. com.apress.todo.repository.ToDoRepository.java

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends
                              CrudRepository<ToDo,String> {}
```

В листинге 5.4 приведен интерфейс `ToDoRepository`, расширяющий `CrudRepository`. Создавать конкретный класс или реализовывать что-либо не нужно; Spring Data JPA реализует его вместо нас. Действия CRUD охватывают все, что необходимо для сохранения данных. Вот и все — можно использовать `ToDoRepository` там, где нужно, без каких-либо дополнительных усилий.

Модель предметной области: класс ToDo

Для использования JPA в полном соответствии с ним необходимо объявить сущность (`@Entity`) и первичный ключ (`@Id`) из модели предметной области. Модифицируем класс `ToDo`, добавив следующие аннотации и методы (листинг 5.5).

Листинг 5.5. com.apress.todo.domain.Todo.java

```
package com.apress.todo.domain;

import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;

@Entity
@Data
@NoArgsConstructor
public class ToDo {

    @NotNull
    @Id
        @GeneratedValue(generator = "system-uuid")
        @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    @NotNull
    @NotBlank
    private String description;
```

```
    @Column(insertable = true, updatable = false)
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;
    public ToDo(String description){
        this.description = description;
    }

    @PrePersist
         void onCreate() {
        this.setCreated(LocalDateTime.now());
        this.setModified(LocalDateTime.now());
    }

    @PreUpdate
        void onUpdate() {
        this.setModified(LocalDateTime.now());
    }
}
```

В листинге 5.5 приведена модифицированная версия модели предметной области `ToDo`. В этом классе появились новые элементы.

- Аннотация `@NoArgsConstructor`. Эта аннотация входит в состав библиотеки Lombok. Она создает конструктор класса без аргументов. В JPA должен быть конструктор без аргументов.
- Аннотация `@Entity`. Указывает, что класс является сущностью и сохраняется в выбранной СУБД.
- Аннотация `@Id`. Задает первичный ключ сущности. Снабженное этой аннотацией поле должно относиться к простому типу данных Java или адаптеру для простого типа данных.
- Аннотация `@GeneratedValue`. Определяет стратегии генерации значений первичного ключа (только для простых, несоставных ключей). Обычно оно используется вместе с аннотацией `@Id`. Существует несколько различных стратегий (`IDENTITY`, `AUTO`, `SEQUENCE` и `TABLE`) и генератор ключей. В данном случае класс определяет в качестве генератора `"system-uuid"` (генерирует уникальный 36-символьный идентификатор).
- Аннотация `@GenericGenerator`. Эта составная часть Hibernate позволяет использовать стратегию для генерации уникального ID из предыдущей аннотации.
- Аннотация `@Column`. Задает отображаемый в сохраняемое свойство столбец; если для поля отсутствует аннотация `@Column`, в качестве значения по умолчанию используется название столбца таблицы базы данных.

- Аннотация `@PrePersist`. Эта аннотация представляет собой обратный вызов, запускаемый до всех действий, связанных с сохранением. Задает новую метку даты/времени для созданных и модифицированных полей до вставки записи в базу данных.
- Аннотация `@PreUpdate`. Эта аннотация представляет собой еще один обратный вызов, запускаемый до всех действий, связанных с обновлением данных. Задает новую метку даты/времени для модифицированного поля до его обновления в базе данных.

Последние две аннотации (`@PrePersist` и `@PreUpdate`) представляют собой очень удобный способ работы с метками даты/времени, упрощая работу разработчика.

Прежде чем читать дальше, проанализируйте отличия кода от предыдущих версий класса модели предметной области ToDo.

Контроллер: класс ToDoController

Настало время модифицировать класс `ToDoController` (листинг 5.6).

Листинг 5.6. com.apress.todo.controller.ToDoController.java

```
package com.apress.todo.controller;

import com.apress.todo.domain.ToDo;
import com.apress.todo.domain.ToDoBuilder;
import com.apress.todo.repository.ToDoRepository;
import com.apress.todo.validation.ToDoValidationError;
import com.apress.todo.validation.ToDoValidationErrorBuilder;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.*;
import org.springframework.web.servlet.support.ServletUriComponentsBuilder;

import javax.validation.Valid;
import java.net.URI;
import java.util.Optional;

@RestController
@RequestMapping("/api")
public class ToDoController {

    private ToDoRepository toDoRepository;
```

```
@Autowired
public ToDoController(ToDoRepository toDoRepository) {
    this.toDoRepository = toDoRepository;
}

@GetMapping("/todo")
public ResponseEntity<Iterable<ToDo>> getToDos(){
    return ResponseEntity.ok(toDoRepository.findAll());
}

@GetMapping("/todo/{id}")
public ResponseEntity<ToDo> getToDoById(@PathVariable String id){
    Optional<ToDo> toDo = toDoRepository.findById(id);
            if(toDo.isPresent())
                return ResponseEntity.ok(toDo.get());

            return ResponseEntity.notFound().build();
}

@PatchMapping("/todo/{id}")
public ResponseEntity<ToDo> setCompleted(@PathVariable String id){
    Optional<ToDo> toDo = toDoRepository.findById(id);
    if(!toDo.isPresent())
        return ResponseEntity.notFound().build();
    ToDo result = toDo.get();
    result.setCompleted(true);
    toDoRepository.save(result);
    URI location = ServletUriComponentsBuilder.fromCurrentRequest().
        buildAndExpand(result.getId()).toUri();
     return ResponseEntity.ok().header("Location",location.toString()).
        build();
}

 @RequestMapping(value="/todo", method = {RequestMethod.
     POST,RequestMethod.PUT})
 public ResponseEntity<?> createToDo(@Valid @RequestBody ToDo toDo,
                                     Errors errors){
    if (errors.hasErrors()) {
         return ResponseEntity.badRequest().
             body(ToDoValidationErrorBuilder.fromBindingErrors(errors));
    }
    ToDo result = toDoRepository.save(toDo);
    URI location = ServletUriComponentsBuilder.fromCurrentRequest().
         path("/{id}").buildAndExpand(result.getId()).toUri();
    return ResponseEntity.created(location).build();
}

@DeleteMapping("/todo/{id}")
public ResponseEntity<ToDo> deleteToDo(@PathVariable String id){
    toDoRepository.delete(ToDoBuilder.create().withId(id).build());
    return ResponseEntity.noContent().build();
}
```

```
    @DeleteMapping("/todo")
    public ResponseEntity<ToDo> deleteToDo(@RequestBody ToDo toDo){
        toDoRepository.delete(toDo);
        return ResponseEntity.noContent().build();
    }

    @ExceptionHandler
    @ResponseStatus(value = HttpStatus.BAD_REQUEST)
    public ToDoValidationError handleException(Exception exception) {
        return new ToDoValidationError(exception.getMessage());
    }
}
```

В листинге 5.6 приведен модифицированный класс `ToDoController`. Теперь он напрямую использует интерфейс `ToDoRepository`, а некоторые из его методов, например `findById`, возвращают тип `Optional` Java 8.

Прежде чем читать дальше, проанализируйте отличия кода от предыдущих версий. Большая часть кода осталась такой же.

Свойства Spring Boot JPA

Spring Boot предоставляет свойства, позволяющие переопределять значения по умолчанию при использовании Spring Data JPA. Одно из них позволяет создавать команды DDL (data definition language, язык описания данных) — возможность, отключенная по умолчанию, которую можно включить для обратного проектирования модели предметной области. Другими словами, это свойство генерирует таблицы и другие связи по классам модели предметной области.

Можно также давать поставщику JPA команды создавать, удалять, обновлять и проверять существующие команды DDL/данные, что удобно при миграции последних. Можно также задать свойство для отображения выполняемых в СУБД операторов SQL.

Добавьте необходимые свойства в файл `application.properties`, как показано в листинге 5.7.

Листинг 5.7. src/main/resources/application.properties

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
```

```
spring.jpa.show-sql=true

# H2
spring.h2.console.enabled=true
```

В листинге 5.7 приведен файл `application.properties` и свойства JPA. В результате этих настроек генерируется таблица на основе класса модели предметной области и создаются таблицы при каждом запуске приложения. Вот список допустимых значений свойства `spring.jpa.hibernate.ddl-auto`:

- `create` — создает схему и удаляет предыдущие данные;
- `create-drop` — создает, а затем и удаляет схему в конце сеанса;
- `update` — обновляет схему при необходимости;
- `validate` — проверяет схему без внесения каких-либо изменений в базе данных;
- `none` — отключает обработку DDL.

Запуск и тестирование: приложение ToDo

Пришло время запустить и протестировать приложение ToDo. Можете запустить его в своей IDE. Если же вы используете Maven — выполните следующее:

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

Если вы используете Gradle, выполните:

```
./gradlew bootRun
```

Для тестирования приложения ToDo можете запустить свое приложение ToDoClient. Оно должно работать без всяких проблем. Можете также отправить запланированные дела с помощью команды `cURL` и посмотреть на результат в консоли H2 (http://localhost:8080/h2-console).

Spring Data REST

Проект Spring Data REST — надстройка над репозиториями Spring Data. Он анализирует классы модели предметной области и делает доступными гипермедийные HTTP-ресурсы с помощью HATEOAS («Гипермедиа как движущая сила состояния приложения», Hypermedia as the Engine of

Application State, HAL +JSON). Вот некоторые из предоставляемых им возможностей.

- Делает доступным REST API из классов модели предметной области с HAL в качестве типа мультимедиа.
- Поддерживает постраничный вывод и делает доступными классы предметной области в виде коллекций.
- Предоставляет специализированные поисковые ресурсы для описанных в репозиториях методов запросов.
- Поддерживает широкие возможности настройки пользовательских контроллеров на случай, если нужно расширить возможности по умолчанию.
- Позволяет включаться в обработку REST-запросов с помощью обработки событий `ApplicationEvent` Spring.
- Обеспечивает отображение браузером HAL всех метаданных; очень удобно при разработке.
- Поддерживает Spring Data JPA, Spring Data MongoDB, Spring Data Neo4j, Spring Data Solr, Spring Data Cassandra и Spring Data Gemfire.

Spring Data REST и Spring Boot

Для использования Spring Data REST в обычном приложении Spring MVC необходимо инициировать его настройку, включив класс `RepositoryRestMvcConfiguration` с помощью аннотации `@Import` в классе JavaConfig (в том, где находится аннотация `@Configuration`); но при непосредственном использовании Spring Boot ничего этого делать не нужно. Spring Boot берет на себя все эти заботы благодаря аннотации `@EnableAutoConfiguration`.

Для использования Spring Data REST в приложении Spring Boot необходимо включить в него зависимости для технологий `spring-boot-starter-data-rest` и `spring-boot-starter-data-*` и/или SQL-драйвер, если вы собираетесь задействовать SQL СУБД.

Приложение ToDo с Spring Data JPA и Spring Data REST

Можете создать приложение ToDo с нуля или просто посмотреть на нужные классы, а также необходимые зависимости в файлах `pom.xml` и `build.gradle`.

Если вы начинаете с нуля, перейдите в браузер и откройте Spring Initializr. Внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-rest`.
- Name (Название): `todo-rest`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Web`, `Lombok`, `JPA`, `REST Repositories`, `H2`, `MySQL`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами IDE (рис. 5.5).

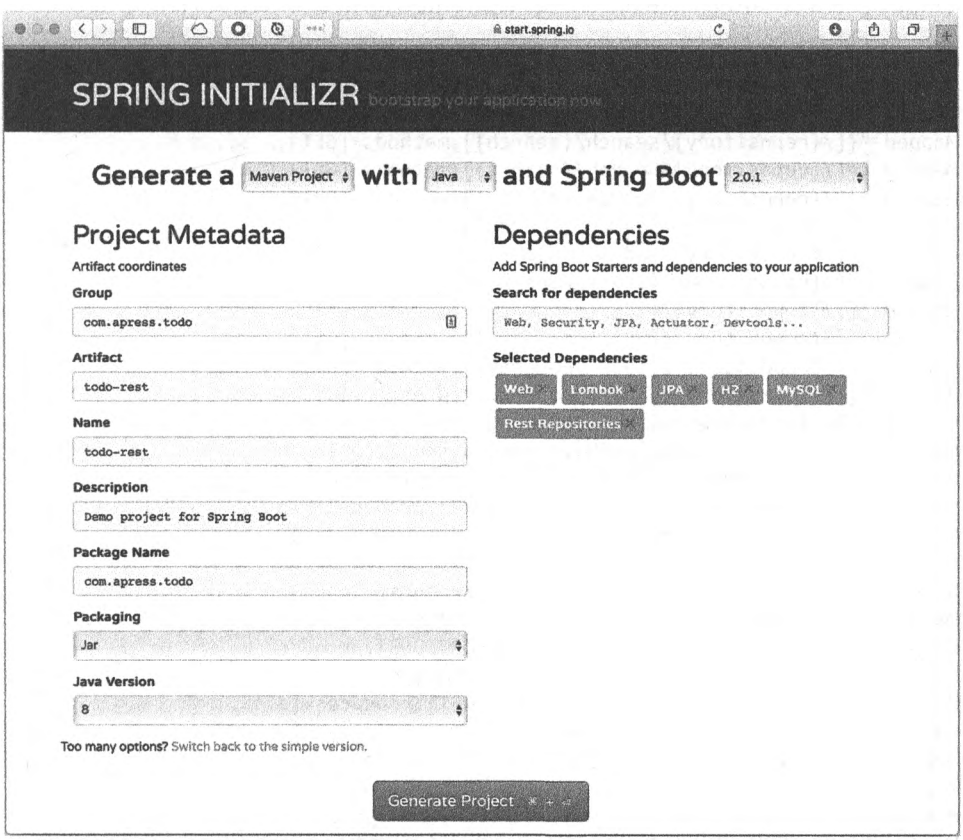


Рис. 5.5. https://start.spring.io

Можете скопировать только классы модели предметной области `ToDo`, `ToDoRepository` и файл `application.properties`; да, только два класса и один файл свойств.

Запуск: приложение ToDo

Пришло время запустить и протестировать приложение ToDo. Можете запустить его в своей IDE. Если вы используете Maven, то выполните:

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

Если вы используете Gradle, выполните:

```
./gradlew bootRun
```

Обратите внимание на выводимые при запуске приложения ToDo журналы:

```
Mapped "{[/{repository}/search],methods=[HEAD],produces  ...
Mapped "{[/{repository}/search],methods=[GET],produces=  ...
Mapped "{[/{repository}/search],methods=[OPTIONS],produ  ...
Mapped "{[/{repository}/search/{search}],methods=[GET],  ...
Mapped "{[/{repository}/search/{search}],methods=[GET],  ...
Mapped "{[/{repository}/search/{search}],methods=[OPTIO  ...
Mapped "{[/{repository}/search/{search}],methods=[HEAD]  ...
Mapped "{[/{repository}/{id}/{property}],methods=[GET],  ...
Mapped "{[/{repository}/{id}/{property}/{propertyId}],m  ...
Mapped "{[/{repository}/{id}/{property}],methods=[DELET  ...
Mapped "{[/{repository}/{id}/{property}],methods=[GET],  ...
Mapped "{[/{repository}/{id}/{property}],methods=[PATCH  ...
Mapped "{[/{repository}/{id}/{property}/{propertyId}],m  ...
Mapped "{[/ || ],methods=[OPTIONS],produces=[applicatio  ...
Mapped "{[/ || ],methods=[HEAD],produces=[application/h  ...
Mapped "{[/ || ],methods=[GET],produces=[application/ha  ...
Mapped "{[/{repository}],methods=[OPTIONS],produces=[ap  ...
Mapped "{[/{repository}],methods=[HEAD],produces=[appli  ...
Mapped "{[/{repository}],methods=[GET],produces=[applic  ...
Mapped "{[/{repository}],methods=[GET],produces=[applic  ...
Mapped "{[/{repository}],methods=[POST],produces=[appli  ...
Mapped "{[/{repository}/{id}],methods=[OPTIONS],produce  ...
Mapped "{[/{repository}/{id}],methods=[HEAD],produces=[  ...
Mapped "{[/{repository}/{id}],methods=[GET],produces=[a  ...
Mapped "{[/{repository}/{id}],methods=[PUT],produces=[a  ...
Mapped "{[/{repository}/{id}],methods=[PATCH],produces=  ...
Mapped "{[/{repository}/{id}],methods=[DELETE],produces  ...
Mapped "{[/profile/{repository}],methods=[GET],produces  ...
Mapped "{[/profile/{repository}],methods=[OPTIONS],prod  ...
Mapped "{[/profile/{repository}],methods=[GET],produces  ...
```

Здесь происходит все задание соответствий конечных точек репозиториев (в этом приложении репозиторий только один) и всех доступных для использования методов HTTP.

Тестирование: приложение ToDo

Для тестирования приложения ToDo мы воспользуемся командой `cURL` и браузером. Приложение ToDoClient пришлось бы модифицировать так, чтобы оно принимало тип мультимедиа HAL+JSON; поэтому использовать его в этом разделе мы не станем. Сначала зайдите в браузер. Перейдите по адресу http://localhost:8080 — и вы должны увидеть что-то вроде изображенного на рис. 5.6.

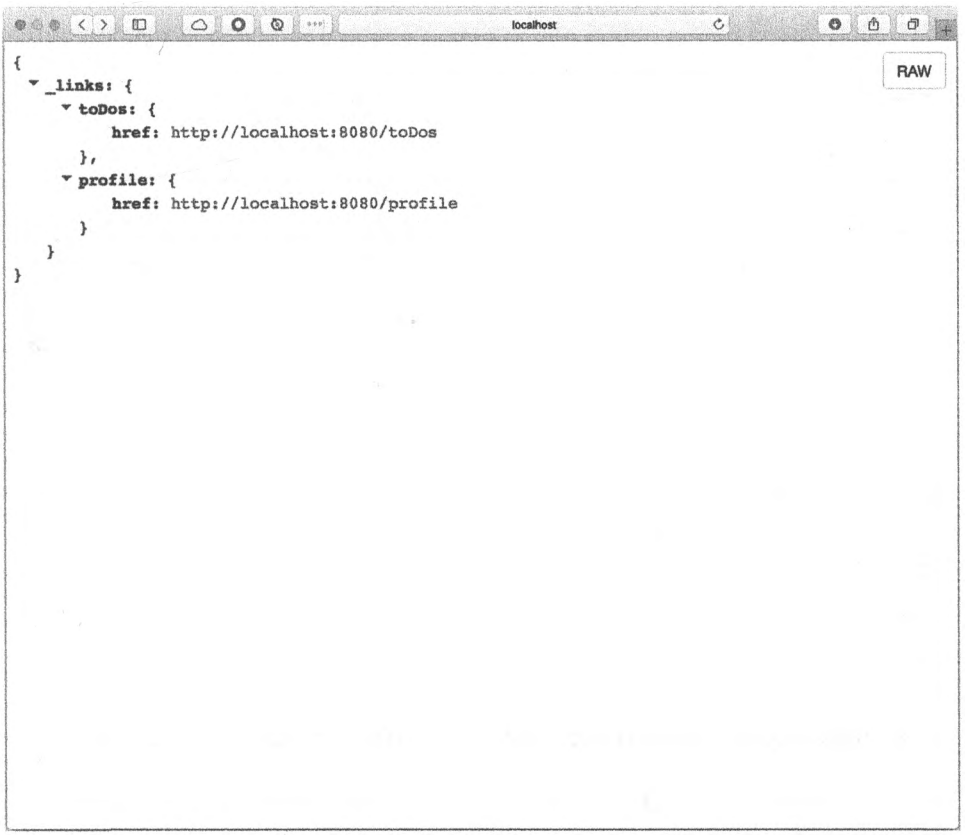


Рис. 5.6. http://localhost:8080

Если вы видите ту же информацию, но в неформатированном виде, попробуйте установить плагин просмотра JSON для браузера и перезагрузите страницу. Он обеспечивает доступность URL http://localhost:8080/toDos в качестве конечной точки, в результате чего можно обращаться к этому URL и выполнять все HTTP-методы (из вышеупомянутых журналов).

Добавим несколько запланированных дел с помощью команды cURL (учтите, вся эта команда представляет собой одну строку).

```
curl -i -X POST -H "Content-Type: application/json" -d '{ "description":"Read
the Pro Spring Boot 2nd Edition Book"}' http://localhost:8080/toDos

HTTP/1.1 201
Location: http://localhost:8080/toDos/8a8080876338ae4e016338b2e2ee0000
Content-Type: application/hal+json;charset=UTF-8
Transfer-Encoding: chunked
Date: Mon, 07 May 2018 03:43:57 GMT

{
  "description" : "Read the Pro Spring Boot 2nd Edition Book",
  "created" : "2018-05-06T21:43:57.676",
  "modified" : "2018-05-06T21:43:57.677",
  "completed" : false,
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/toDos/8a8080876338ae4e016338b2e2ee0000"
    },
    "toDo" : {
      "href" : "http://localhost:8080/toDos/8a8080876338ae4e016338b2e2ee0000"
    }
  }
}
```

Вы получили результат в виде HAL+JSON. После добавления еще нескольких запланированных дел можете вернуться в браузер и щелкнуть на ссылке http://localhost:8080/toDos. Вы должны увидеть нечто напоминающее рис. 5.7.

На рис. 5.7 показан HAL+JSON ответ при обращении к конечной точке http://localhost:8080/toDos.

Тестирование с помощью браузера HAL: приложение ToDo

В проекте Spring Data REST есть утилита — браузер HAL. Он представляет собой веб-приложение, с помощью которого разработчики могут интерактивно визуализировать конечные точки. Так что если вы не хотите применять

Рис. 5.7. http://localhost:8080/toDos

непосредственно конечные точки и/или команды cURL, можете воспользоваться браузером HAL.

Для использования браузера HAL необходимо добавить следующую зависимость. Если вы используете Maven, добавьте следующее в свой файл `pom.xml`:

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-rest-hal-browser</artifactId>
</dependency>
```

Если вы используете Gradle, добавьте следующее в свой файл `build.gradle`:

```
compile 'org.springframework.data:spring-data-rest-hal-browser'
```

Теперь можете перезапустить приложение ToDo и перейти в браузере прямо по адресу http://localhost:8080. Вы увидите то же, что показано на рис. 5.8.

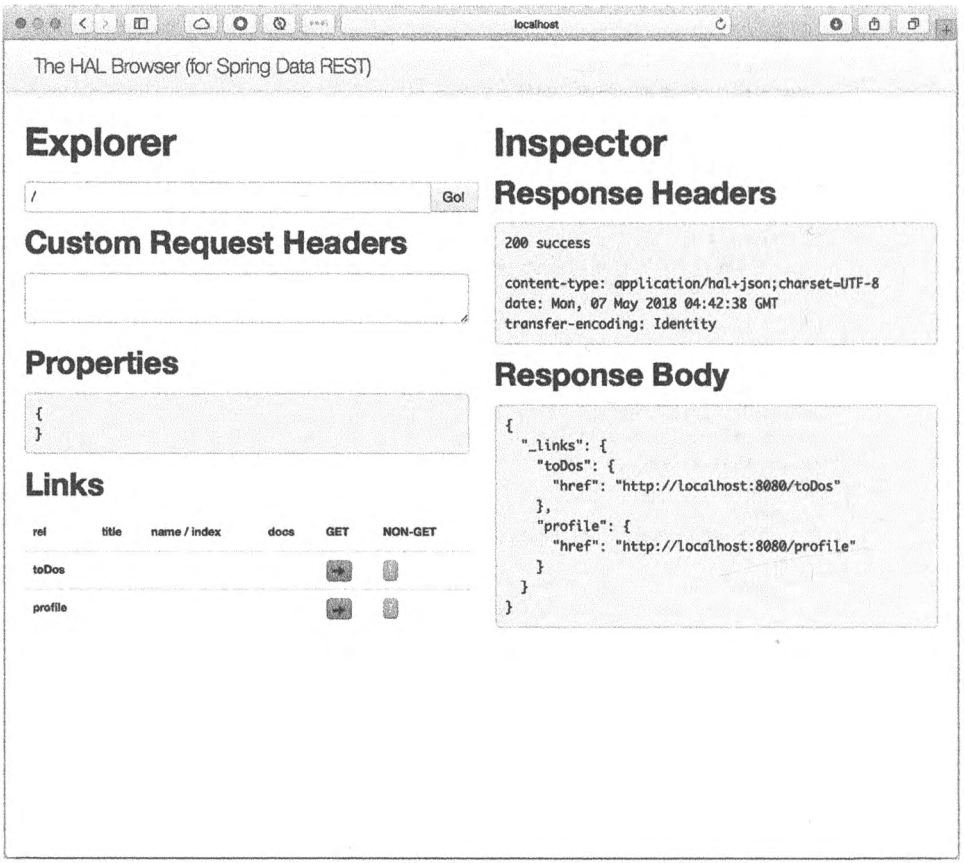


Рис. 5.8. http://localhost:8080

Для работы с каждой из отдельных конечных точек и каждым из HTTP-методов можете щелкнуть на картинках в столбцах GET и NON-GET. Для разработчиков это очень удобно.

Я показал вам лишь часть из множества возможностей Spring Data REST. Никакой контроллер, по сути, больше не нужен. Кроме того, можно сделать доступным любой класс предметной области благодаря легкости переопределения в Spring Boot и Spring Data REST.

Базы данных NoSQL

Базы данных NoSQL — еще один вариант хранения данных, отличающийся от реляционных баз данных. Существует классификация этих новых баз данных NoSQL в соответствии с их моделью данных.

- Столбцовые (Cassandra, HBase и т. д.).
- Документоориентированные (CouchDB, MongoDB и т. д.).
- Хранилища пар «ключ/значение» (Redis, Riak и т. д.).
- Графовые (Neo4J, Virtuoso и т. д.).
- Мультимодельные (OrientDB, ArangoDB и т. д.).

Как вы видите, вариантов множество. Мне кажется, что главное в базе данных — масштабируемость и способность обрабатывать миллионы записей.

Spring Data MongoDB

Проект Spring Data MongoDB позволяет взаимодействовать с документоориентированной базой данных MongoDB. При этом вы, что важно, продолжаете работать с классами модели предметной области, которые используют аннотацию `@Document`, и объявлять интерфейсы, применяющие `CrudRepository<T,ID>`. В результате получается необходимый MongoDB для сохранения данных набор.

Вот некоторые из возможностей этого проекта.

- Spring Data MongoDB предоставляет вспомогательный класс `MongoTemplate` (очень похожий на `JdbcTemplate`), берущий на себя все стереотипные взаимодействия с документоориентированной базой данных MongoDB.
- События жизненного цикла сохранения и отображения.
- Вспомогательный класс `MongoTemplate`, а также низкоуровневые отображения с помощью абстракций `MongoReader/MongoWriter`.
- Основанный на Java DSL для запросов, задания поисковых критериев и обновления.
- Поддержка геопространственных данных, интеграция с MapReduce и поддержка GridFS.

- Поддержка персистентности данных с использованием нескольких хранилищ для сущностей JPA. Это означает возможность задействовать классы, снабженные аннотацией `@Entity` и другими аннотациями, в том числе для сохранения/извлечения данных с помощью документоориентированной базы данных MongoDB.

Использование Spring Data MongoDB со Spring Boot

Для использования MongoDB со Spring Boot необходима возможность доступа к экземпляру сервера MongoDB, а также нужно добавить зависимость `spring-boot-starter-data-mongodb`.

Spring Boot использует возможности автоконфигурации для настройки взаимодействия с экземпляром сервера MongoDB. По умолчанию Spring Boot пытается подключиться к локальной машине на порту 27017 (стандартный порт MongoDB). К удаленному серверу MongoDB можно подключиться с помощью переопределения настроек по умолчанию. Для этого необходимо воспользоваться свойствами `spring.mongodb.*` в файле `application.properties` (более простой способ) или объявить компонент с помощью XML или в классе JavaConfig.

Spring Boot также автоматически конфигурирует класс `MongoTemplate` (очень похожий на `JdbcTemplate`), а потому готов к любым взаимодействиям с сервером MongoDB. Еще одна замечательная возможность — работа с *репозиториями*, то есть возможность повторного использования того же интерфейса, что и для JPA.

Установка MongoDB

Прежде всего нужно, чтобы у вас на машине был установлен сервер MongoDB.

Если вы работаете в Mac/Linux, где есть команда `brew` (http://brew.sh/), выполните следующую команду:

```
brew install mongodb
```

Запустить его можно с помощью команды:

```
mongod
```

Или можете установить MongoDB, скачав его с веб-сайта https://www.mongodb.org/downloads#production и следуя прилагаемым инструкциям.

Встроенная MongoDB

Существует и другой способ применения MongoDB, по крайней мере в среде разработки. А именно: можно воспользоваться встроенной MongoDB (MongoDB Embedded). Обычно она применяется в среде тестирования, но можно легко запустить ее в режиме разработки с областью видимости runtime.

Для использования встроенного MongoDB необходимо добавить следующую зависимость. Если вы применяете Maven, добавьте ее в файл `pom.xml`:

```
<dependency>
      <groupId>de.flapdoodle.embed</groupId>
      <artifactId>de.flapdoodle.embed.mongo</artifactId>
      <scope>runtime</scope>
</dependency>
```

Если вы используете Gradle, добавьте следующее:

```
runtime('de.flapdoodle.embed:de.flapdoodle.embed.mongo')
```

Далее необходимо настроить клиент Mongo на использование встроенного сервера MongoDB (листинг 5.8).

Листинг 5.8. com.apress.todo.config.ToDoConfig.java

```
package com.apress.todo.config;

import com.mongodb.MongoClient;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.DependsOn;
import org.springframework.core.env.Environment;

@Configuration
public class ToDoConfig {

    private Environment environment;
    public ToDoConfig(Environment environment){
        this.environment = environment;
    }
```

```
    @Bean
        @DependsOn("embeddedMongoServer")
    public MongoClient mongoClient() {
        int port =
            this.environment.getProperty("local.mongo.port", Integer.class);
        return new MongoClient("localhost",port);
    }
}
```

В листинге 5.8 приведена конфигурация компонента `MongoClient`. Встроенный MongoDB использует выбираемый случайным образом порт при запуске приложения, поэтому необходимо также задействовать компонент `Environment`.

При таком подходе к применению сервера MongoDB никаких других свойств задавать не нужно.

Приложение ToDo с использованием Spring Data MongoDB

Можете создать приложение ToDo с нуля или же просто посмотреть на нужные классы, а также необходимые зависимости в файлах `pom.xml` и `build.gradle`.

Если вы начинаете с нуля, перейдите в браузер и откройте Spring Initializr. Внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-mongo`.
- Name (Название): `todo-mongo`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Web`, `Lombok`, `MongoDB`, `Embedded MongoDB`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами IDE (рис. 5.9).

Можете скопировать все классы из проекта todo-jpa. В следующем разделе вы увидите, какие из них необходимо модифицировать.

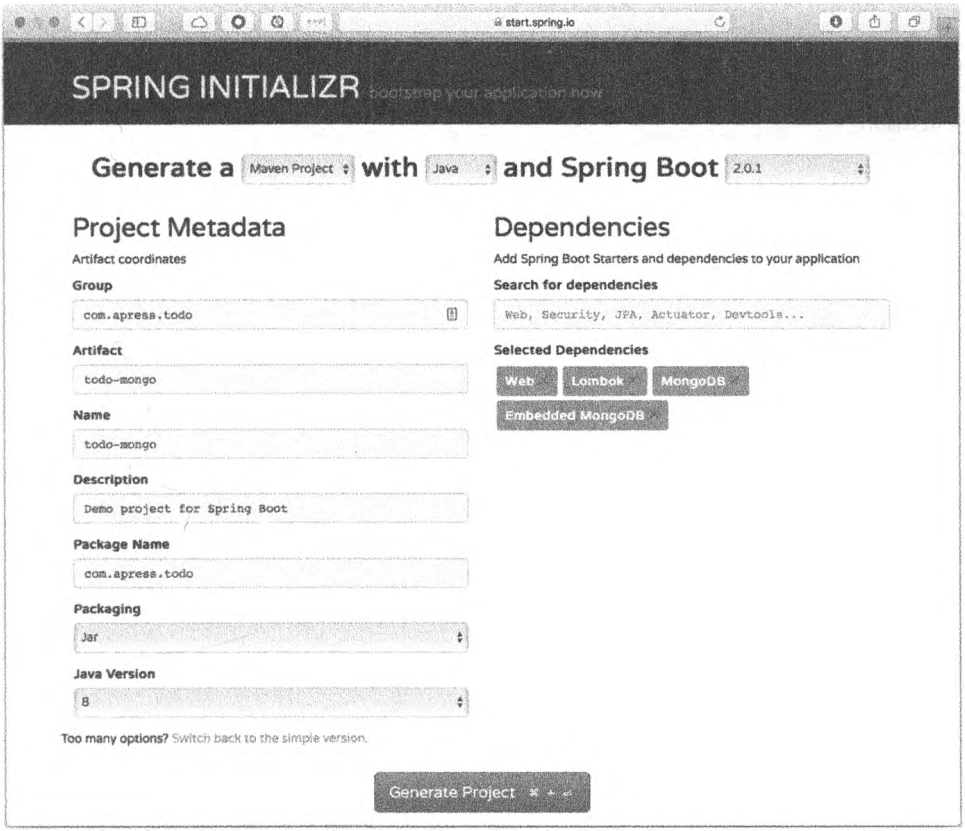


Рис. 5.9. https://start.spring.io

Модель предметной области: ToDo

Откройте класс модели предметной области `ToDo` и отредактируйте его в соответствии с листингом 5.9.

Листинг 5.9. com.apress.todo.domain.ToDo.java

```
package com.apress.todo.domain;

import lombok.Data;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import javax.validation.constraints.NotBlank;
```

```
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;
import java.util.UUID;

@Document
@Data
public class ToDo {
    @NotNull
    @Id
    private String id;
    @NotNull
    @NotBlank
    private String description;

    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
        LocalDateTime date = LocalDateTime.now();
        this.id = UUID.randomUUID().toString();
        this.created = date;
        this.modified = date;
    }

    public ToDo(String description){
        this();
        this.description = description;
    }
}
```

В листинге 5.9 показан модифицированный класс `ToDo`. Он снабжен аннотацией `@Document`, указывающей на постоянное хранение данных, а также аннотацией `@Id` для объявления уникального ключа.

Если вы используете удаленный сервер MongoDB, то можете переопределить настройки по умолчанию, указывающие на локальную машину. Перейдите в этом случае в файл `application.properties` и добавьте следующие свойства:

```
## MongoDB
spring.data.mongodb.host=my-remote-server
spring.data.mongodb.port=27017
spring.data.mongodb.username=myuser
spring.data.mongodb.password=secretpassword
```

Теперь можете снова посмотреть на классы `ToDoRepository` и `ToDoController`, которые вообще менять не нужно. Spring Data удобен возможностью переис-

пользования модели для хранения данных с применением нескольких хранилищ и переиспользования всех предыдущих классов, что сильно ускоряет и упрощает разработку.

Запуск и тестирование: приложение ToDo

Пришло время запустить и протестировать приложение ToDo. Можете запустить его в своей IDE или, если вы используете Maven, выполните следующее:

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

Если вы используете Gradle, выполните:

```
./gradlew bootRun
```

Для тестирования приложения ToDo можете запустить свое приложение ToDoClient — и этого достаточно. Переход с одной системы постоянного хранения на другую не составляет никаких сложностей. Наверное, вам интересно, что будет, если вы захотите воспользоваться MapReduce или низкоуровневыми операциями. Что ж, это возможно с помощью класса `MongoTemplate`.

Приложение ToDo со Spring Data MongoDB REST

Как создать приложение MongoDB REST? Достаточно просто добавить зависимость `spring-boot-starter-data-rest` в свой файл `pom.xml` или `build.gradle`! Конечно, необходимо удалить пакеты для контроллера и проверки корректности данных, а также класс `ToDoBuilder` — достаточно только двух классов.

Помните, что Spring Data REST делает доступными конечные точки репозитория, а в качестве типа мультимедиа использует HATEOAS (HAL+JSON).

ПРИМЕЧАНИЕ

Программное решение для этого раздела можно найти в прилагаемом к книге исходном коде на веб-сайте Apress или в GitHub по адресу https://github.com/Apress/pro-spring-boot-2. Проект называется todo-mongo-rest.

Spring Data Redis

Проект Spring Data Redis позволяет легко настраивать сервер Redis и обращаться к нему. В него включены как низкоуровневые, так и высокоуровневые абстракции для взаимодействия с Redis, и он предоставляет класс `RedisTemplate` и постоянное хранение данных на базе репозиториев, следуя тем же стандартам Spring Data.

Вот некоторые из возможностей Spring Data Redis.

- Класс `RedisTemplate` играет роль высокоуровневой абстракции для всех операций Redis.
- Обмен сообщениями посредством механизма публикации/подписки.
- Поддержка Redis Sentinel и Redis Cluster.
- Возможность использования пакетов для подключения на низком уровне с различными драйверами, например Jedis и Lettuce.
- Репозитории, сортировка и постраничный вывод с помощью аннотации `@EnableRedisRepositories`.
- Redis реализует абстракцию кэширования Spring, так что можно использовать Redis в качестве механизма веб-кэширования.

Использование Spring Data Redis со Spring Boot

Чтобы использовать Spring Data Redis, необходимо лишь добавить зависимость `spring-boot-starter-data-redis` для доступа к серверу Redis.

Spring Boot для задания настроек по умолчанию сервера Redis использует автоконфигурацию. При применении репозитория автоматически задействуется аннотация `@EnableRedisRepositories` (ее можно не добавлять).

По умолчанию используются локальная машина и порт 6379. Конечно, эти настройки по умолчанию можно переопределить, меняя значения свойств `spring.redis.*` в файле `application.properties`.

Приложение ToDo со Spring Data Redis

Можете создать приложение ToDo с нуля или просто посмотреть на нужные классы, а также необходимые зависимости в файлах `pom.xml` и `build.gradle`.

Если вы начинаете с нуля, перейдите в браузер и откройте Spring Initializr. Внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-redis`.
- Name (Название): `todo-redis`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Web`, `Lombok`, `Redis`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами IDE (рис. 5.10).

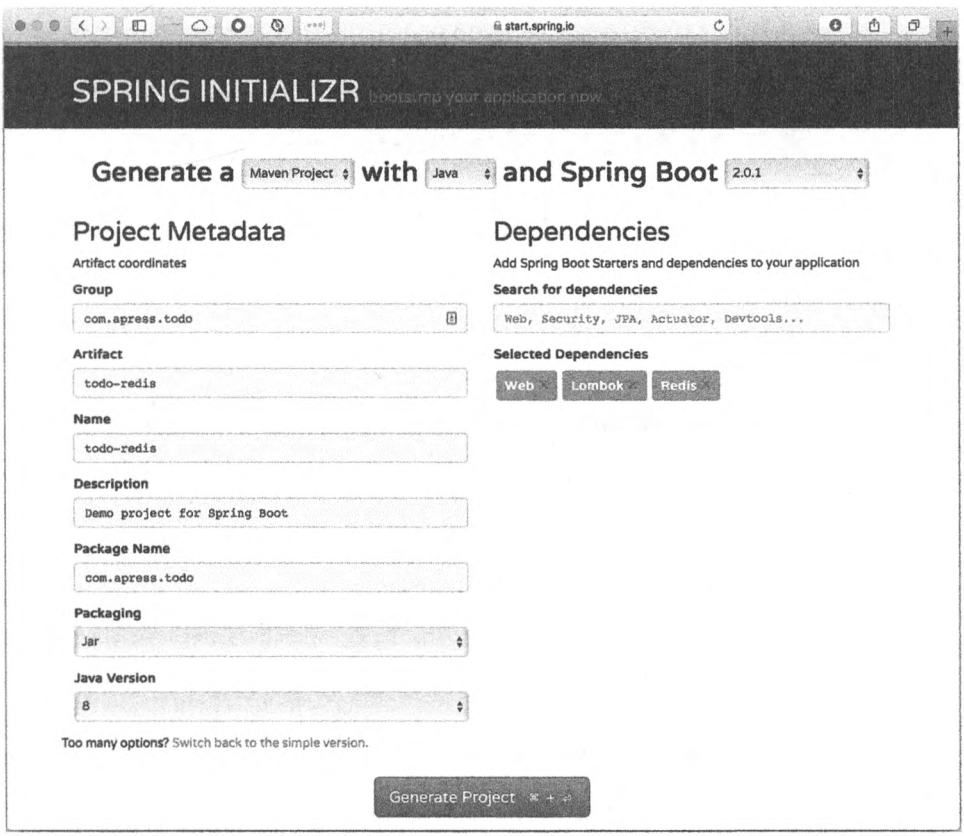


Рис. 5.10. https://start.spring.io

Можете скопировать все классы из проекта todo-mongo. В следующем разделе вы увидите, какие из них необходимо модифицировать.

Модель предметной области: ToDo

Откройте класс модели предметной области ToDo и отредактируйте его в соответствии с листингом 5.10.

Листинг 5.10. com.apress.todo.ToDo.java

```
package com.apress.todo.domain;

import lombok.Data;
import org.springframework.data.annotation.Id;
import org.springframework.data.redis.core.RedisHash;

import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;
import java.util.UUID;

@Data
@RedisHash
public class ToDo {

    @NotNull
    @Id
    private String id;
    @NotNull
    @NotBlank
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
        LocalDateTime date = LocalDateTime.now();
        this.id = UUID.randomUUID().toString();
        this.created = date;
        this.modified = date;
    }

    public ToDo(String description){
        this();
        this.description = description;
    }
}
```

В листинге 5.10 приведен единственный класс, который поменялся. Он снабжен аннотацией `@RedisHash`, которая помечает класс как персистентный, а также аннотацией `@Id` в качестве части составного ключа. Хеш при вставке объекта `ToDo` содержит ключ в формате `class:id`. Для этого приложения составной ключ выглядит примерно следующим образом: `"com.apress.todo.domain.ToDo: bbee6e32-37f3-4d5a-8f29-e2c79a28c301"`.

Если вы используете удаленный сервер Redis, то можете переопределить настройки по умолчанию, указывающие на локальную машину. Перейдите в этом случае в файл `application.properties` и добавьте следующие свойства.

```
## Redis - удаленный сервер
spring.redis.host=my-remote-server
spring.redis.port=6379
spring.redis.password=my-secret
```

Можете взглянуть на классы `ToDoRepository` и `ToDoController`, которые вообще менять не нужно; все осталось таким же, как и раньше.

Запуск и тестирование: приложение ToDo

Пришло время запустить и протестировать приложение ToDo. Можете запустить его в своей IDE. Или, если вы используете Maven, выполните следующее:

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

Если вы используете Gradle, выполните:

```
./gradlew bootRun
```

Для тестирования приложения ToDo можете запустить свое приложение ToDoClient — и этого достаточно. Если же вы оперируете другой структурой (например, `SET`, `LIST`, `STRING`, `ZSET`) для *низкоуровневых* операций, то можете воспользоваться классом `RedisTemplate`, который Spring Boot уже настроил и подготовил для применения.

ПРИМЕЧАНИЕ

Напоминаю, что исходный код для данной книги можно скачать с веб-сайта Apress или из GitHub: https://github.com/Apress/pro-spring-boot-2.

Дополнительные возможности по работе с данными с помощью Spring Boot

Spring Boot поддерживает еще множество возможностей и механизмов для выполнения операций над данными, например DSL jOOQ (Java Object-Oriented Querying, объектно-ориентированные запросы Java, www.jooq.org) для генерации Java-кода на основе базы данных с возможностью создания типобезопасных SQL-запросов с помощью его собственного DSL.

Можно также производить миграцию базы данных с помощью запускаемых в самом начале Flyway (https://flywaydb.org/) или Liquibase (http://www.liquibase.org/).

Несколько источников данных. Одна из важных (я убежден, что совершенно необходимых) возможностей Spring Boot — работа с несколькими экземплярами `DataSource`, независимо от используемой технологии хранения данных.

Как вы уже знаете, Spring Boot обеспечивает автоконфигурацию по умолчанию, на основе пути к классам, причем ее можно переопределить без всяких проблем. Для использования нескольких экземпляров `DataSource`, возможно указывающих на различные базы данных и/или различные СУБД, необходимо переопределить настройки по умолчанию. Если вы помните, в главе 4 мы создали законченное, хотя и простое, веб-приложение, требовавшее определенных настроек: `DataSource`, `EntityManager`, `TransactionManager`, `JpaVendor` и т. д. Для использования нескольких источников данных необходимо добавить такую же конфигурацию. Другими словами, следует добавить в настройки несколько `EntityManager`, `TransactionManager` и т. д.

Как выполнить это для приложения ToDo? Посмотрите снова на то, что мы делали в главе 4. Внимательно изучите все настройки, и вы поймете, что нужно делать.

Решение этой задачи можно найти в исходном коде к данной книге. Соответствующий проект называется `todo-rest-2ds`. Он содержит все классы предметной области `User` и `ToDo`, сохраняющие данные в отдельные базы данных.

Резюме

В этой главе мы изучили различные технологии работы с данными и их взаимодействие со Spring Boot. Мы также узнали, что Spring Boot использует свои возможности автоконфигурации для применения настроек по умолчанию на основе пути к классам.

Вы узнали, что при наличии двух или более SQL-драйверов, один из которых H2, HSQL или Derby, Spring Boot настраивает встроенную СУБД, если предварительно не был описан никакой экземпляр `DataSource`. Я показал вам, как настраивать и переопределять некоторые из настроек по умолчанию для работы с данными. Вы узнали, что Spring Boot реализует паттерн «Шаблонный метод» для скрытия «под капотом» всех сложных задач, обычно необходимых без фреймворка Spring.

В следующей главе мы выйдем на новый уровень работы с веб-технологиями и данными, обсудим реактивное программирование и изучим WebFlux и реактивные данные.

6 Работа с WebFlux и Reactive Data

В этой главе я покажу вам новейший компонент фреймворка Spring 5 и его использование со Spring Boot. Spring WebFlux — новый реактивный стек для веб-приложений. Он появился в версии 5.0 фреймворка Spring и представляет собой полностью неблокирующий фреймворк, в основе которого лежит проект Reactor, поддерживает противодавление для реактивных потоков данных и работает на таких серверах, как Netty и Undertow, а также в контейнерах Servlet 3.1+.

Прежде чем показать вам, как использовать WebFlux со Spring Boot, поговорим о реактивных системах и их реализации проектом Reactor (https://projectreactor.io/).

Реактивные системы

За прошедшее десятилетие мы были свидетелями изменения программного обеспечения, повышения его стабильности, надежности и гибкости по отношению ко все более современным требованиям со стороны не только пользователей (оперирующих веб- и традиционными приложениями), но и множества устройств (мобильных телефонов, датчиков и т. д.). Эти новые требования означают множество новых непростых задач; именно поэтому несколько организаций совместно разработали манифест, охватывающий многие аспекты современных требований к данным.

Манифест реактивных систем

Манифест реактивных систем (https://www.reactivemanifesto.org/ru) был подписан 16 сентября 2014 года. Он определяет, какими должны быть реактивные системы. Реактивные системы — гибкие, слабо связанные и масштабируемые. Они более устойчивы к сбоям и обрабатывают произошедший сбой на основе паттернов, чтобы избежать аварийной ситуации. Эти реактивные системы должны обладать определенными отличительными признаками.

Реактивные системы:

- *быстро реагирующие.* Большинство таких систем реагирует на действия пользователя своевременно, если это возможно; они стараются обеспечить короткое и одинаковое время отклика и надежность в смысле обеспечения стабильного уровня качества обслуживания;
- *отказоустойчивые.* Устойчивость систем обеспечивается за счет использования паттернов репликации, обособленности, изоляции и делегирования. Сбои систем обособляются с помощью изоляции; сбои не должны влиять на другие системы. Восстановление должно производиться на основе другой системы для гарантии высокой доступности (high availability, HA);
- *адаптивные.* Система должна быстро реагировать при любой нагрузке. Реактивные системы способны реагировать на изменения темпов поступления входных данных повышением или уменьшением ресурсов, выделяемых на обработку этих данных. Не должно быть никаких узких мест, то есть система должна уметь совместно использовать или реплицировать компоненты. Реактивные системы обязаны поддерживать прогнозирующие алгоритмы, что обеспечивает экономически оправданный уровень адаптивности на серийном аппаратном обеспечении.
- *ориентированы на обмен сообщениями.* Реактивные системы должны работать на основе асинхронного обмена сообщениями для формирования границ между компонентами, чтобы обеспечить слабую сцепленность, изоляцию и прозрачность размещения систем. Они должны поддерживать адаптивность, управление нагрузкой и потоком событий за счет воплощения при необходимости паттерна противодавления. Обмен сообщениями должен быть неблокирующим, чтобы ресурсы потреблялись только при активной работе, а значит, уменьшались накладные расходы системы.

С появлением Манифеста реактивных систем начали возникать различные инициативы по реализации фреймворков и библиотек, которые могли бы

помочь разработчикам по всему миру. Реактивные потоки данных (Reactive Streams, www.reactive-streams.org) — спецификация, определяющая четыре простых интерфейса (`Publisher<T>` — поставщик неограниченной последовательности элементов, публикующий их согласно потребностям подписчика; `Subscriber<T>` — подписчик, подписывается на издателя; `Subscription` отражает (один к одному) жизненный цикл подписанного на издателя подписчика; и `Processor`, представляющий этап обработки как подписчика, так и издателя) и различные реализации, например ReactiveX RXJava (http://reactivex.io/), Akka Streams (https://akka.io/), Ratpack (https://ratpack.io/), Vert.X (https://vertx.io/), Slick, проект Reactor и др.

API реактивных потоков данных был реализован в SDK Java 9; другими словами, по состоянию на декабрь 2017 года Reactive Streams версии 1.0.2 были частью JDK9.

Project Reactor

Project Reactor 3.0 представляет собой библиотеку, в основе которой лежит спецификация реактивных потоков данных, и привносит парадигму реактивного программирования в JVM. Парадигма *реактивного программирования* (reactive programming) основана на событийной модели, в которой данные передаются потребителю по мере их появления; в ней обрабатываются асинхронные последовательности событий. Реактивное программирование предоставляет полностью асинхронный и неблокирующий паттерны и является альтернативой ограниченным возможностям по созданию асинхронного кода в JDK (функциям обратного вызова, API и интерфейсу `Future<V>`).

Reactor представляет собой полнофункциональный неблокирующий фреймворк реактивного программирования с контролем обратного потока данных и интеграцией взаимодействия с функциональными API Java 8 (`CompletableFuture`, `Stream` и `Duration`). Reactor предоставляет два реактивных, пригодных для компоновки асинхронных API; Flux [N] (для N элементов) и Mono [0|1] (для 0 или 1 элемента). Reactor подходит для разработки микросервисных архитектур, поскольку предоставляет возможности обмена сообщениями между процессами (interprocess communication, IPC) с помощью компонентов *reactor-ipc* и позволяет использовать сетевые механизмы для

HTTP (включая WebSockets, TCP и UDP), а также полностью поддерживает реактивное кодирование/декодирование.

Project Reactor предоставляет узлы-обработчики, операторы и таймеры, позволяющие поддерживать очень высокую пропускную способность в десятки миллионов сообщений в секунду при низком объеме потребляемой памяти.

ПРИМЕЧАНИЕ

Чтобы узнать больше о проекте Reactor, посетите сайт https://projectreactor.io/ и загляните в его документацию по адресу http://projectreactor.io/docs/core/release/reference/docs/index.html.

Создание приложения ToDo с использованием Reactor

Что ж, начнем использовать Reactor в приложении ToDo и поэкспериментируем с API Flux и Mono. В этом разделе мы создадим пример приложения Reactor для работы со списком запланированных дел.

Откройте свой любимый браузер и зайдите на сайт Spring Initializr (https://start.spring.io); внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.reactor`.
- Artifact (Артефакт): `example`.
- Dependencies (Зависимости): `Lombok`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами IDE (рис. 6.1).

Как вы видите, на этот раз никаких существенных зависимостей, кроме Lombok (https://projectlombok.org/), нет. Мы постараемся максимально все упростить. В этом примере мы лишь «попробуем на вкус» API Flux и Mono. А позднее создадим веб-приложение с помощью фреймворка WebFlux.

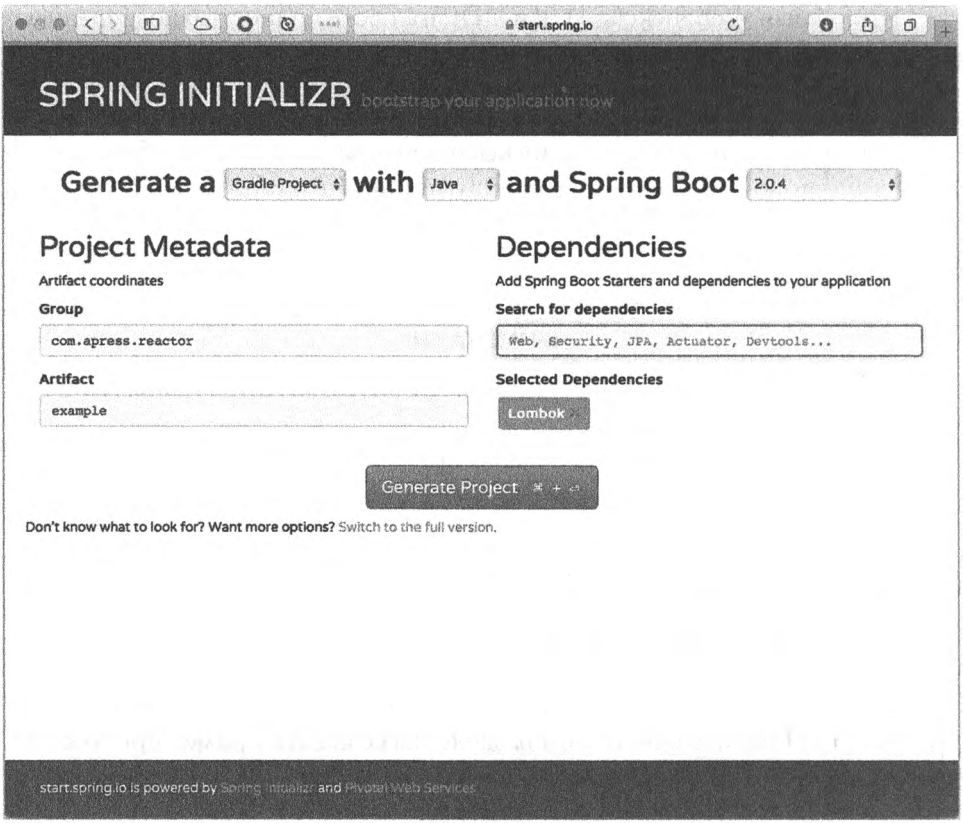


Рис. 6.1. Spring Initializr (https://start.spring.io)

Если вы используете Maven, откройте файл `pom.xml` и добавьте в него следующий раздел и зависимость:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-bom</artifactId>
            <version>Bismuth-SR10</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
<dependencies>
      <!-- ... Прочие зависимости ... -->
      <dependency>
```

```
            <groupId>io.projectreactor</groupId>
            <artifactId>reactor-core</artifactId>
        </dependency>
</dependencies>
```

Reactor обладает транзитивной зависимостью от JAR-файлов `reactive-streams` и предоставляет все необходимые JAR-файлы путем добавления перечня компонентов.

Если вы используете Gradle, добавьте следующий раздел и зависимость в файл `build.gradle`.

```
dependencyManagement {
    imports {
        mavenBom "io.projectreactor:reactor-bom:Bismuth-SR10"
    }
}
dependencies {
      // ... Прочие зависимости ...
      compile('io.projectreactor:reactor-core')
}
```

Создадим класс предметной области `ToDo`, но на этот раз без возможностей постоянного хранения данных (листинг 6.1).

Листинг 6.1. com.apress.reactor.example.domain.ToDo.java

```
package com.apress.reactor.example.domain;

import lombok.Data;

import java.time.LocalDateTime;
@Data
public class ToDo {

    private String id;
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){}
    public ToDo(String description){
        this.description = description;
    }

    public ToDo(String description, boolean completed){
        this.description = description;
        this.completed = completed;
    }
}
```

В листинге 6.1 приведен класс `ToDo`. Ничего особенного в нем нет, но мы ранее работали с технологиями постоянного хранения данных; в этом случае же можно оставить все как есть, не усложняя. Начнем с описания реактивных API Mono и Flux и добавим необходимый для использования класса предметной области `ToDo` код.

Mono<T>, асинхронный результат из [0|1] элемента

`Mono<T>` представляет собой специализированную разновидность интерфейса `Publisher<T>`, которая выдает один элемент, причем может (не обязательно) завершаться с помощью сигнала `onComplete` или `onError`. Для выполнения операций над выданным элементом можно применять различные операторы (рис. 6.2).

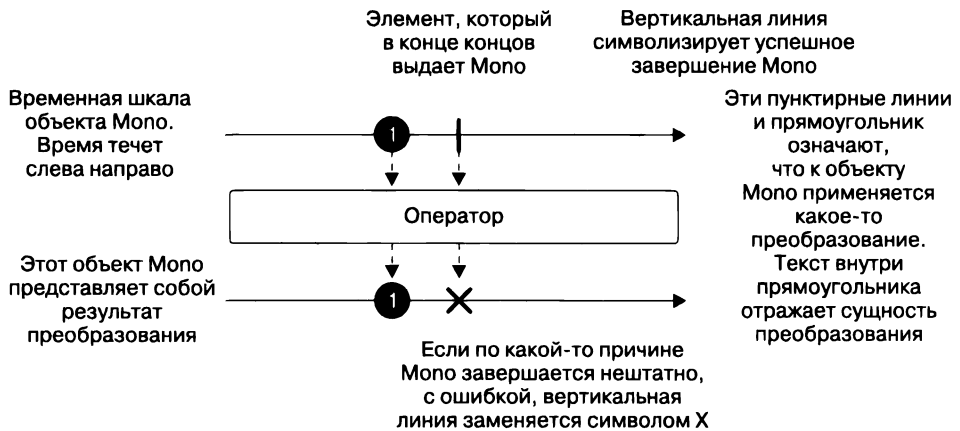


Рис. 6.2. Mono [0 | 1] (изображение взято из документации http://projectreactor.io/)

Создадим класс `MonoExample` и научимся использовать API Mono (листинг 6.2).

Листинг 6.2. com.apress.reactor.example.MonoExample.java

```
package com.apress.reactor.example;

import com.apress.reactor.example.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.Configuration;
import reactor.core.publisher.Mono;
import reactor.core.publisher.MonoProcessor;
import reactor.core.scheduler.Schedulers;

import java.time.Duration;

@Configuration
public class MonoExample {

    static private Logger LOG = LoggerFactory.getLogger(MonoExample.class);
    @Bean
    public CommandLineRunner runMonoExample(){
       return args -> {

       MonoProcessor<ToDo> promise = MonoProcessor.create();
       Mono<ToDo> result = promise
            .doOnSuccess(p -> LOG.info("MONO >> ToDo: {}",
                p.getDescription()))
            .doOnTerminate( () -> LOG.info("MONO >> Done"))
            .doOnError(t -> LOG.error(t.getMessage(), t))
            .subscribeOn(Schedulers.single());

            promise.onNext(new ToDo("Buy my ticket for SpringOne Platform
                2018"));
            //promise.onError(new IllegalArgumentException("There is an error
            //processing the ToDo..."));
            result.block(Duration.ofMillis(1000));
        };
    }
}
```

В листинге 6.2 приведен класс `MonoExample`; обсудим его подробнее.

- `MonoProcessor`. В Reactor существует понятие узлов-обработчиков — разновидностей издателей, являющихся также подписчиками; это значит, что можно не только подписаться на узел-обработчик, но и вызывать методы для внедрения данных в последовательность вручную или для ее завершения. В данном случае мы используем метод `onNext` для выдачи экземпляра `ToDo`.
- `Mono`. Издатель реактивного потока данных, поддерживающий основные операции, который завершается либо успешной выдачей элемента, либо ошибкой.
- `doOnSuccess`. Этот метод вызывается или срабатывает при успешном завершении `Mono`.

- `doOnTerminate`. Этот метод вызывается или срабатывает, когда выполнение `Mono` прерывается либо успешным завершением, либо ошибкой.
- `doOnError`. Этот метод вызывается при завершении `Mono` ошибкой.
- `subscribeOn`. Подписывается на тип `Mono` и запрашивает неограниченные требования на указанного исполнителя (worker) `Scheduler`.
- `onNext`. Этот метод выдает значение, которое может быть помечено аннотацией `@Nullable`.
- `block`. Подписывается на тип `Mono` и осуществляет блокировку до момента получения следующего сигнала или истечения времени ожидания.

Как видите, этот пример очень прост, но не забывайте, что мы говорим о реактивном программировании, в котором вместо блокировок или опросов сервера производится «проталкивание» данных потребителю до тех пор, пока он не вернет сигнал завершения. Благодаря ему приложения становятся более эффективными и надежными. Без преувеличения можно сказать, что медленные потребители остались в прошлом.

Можете теперь запустить приложение с помощью командной строки или своей IDE. Вы должны увидеть следующий вывод:

```
INFO 55588 - [single-1] c.a.r.e.MonoExample   : MONO >> ToDo: Buy my ticket
for SpringOne Platform 2018
INFO 55588 - [single-1] c.a.r.e.MonoExample   : MONO >> Done
```

ПРИМЕЧАНИЕ

Как работает этот код? Помните, что мы снабдили наш класс аннотацией @Configuration и объявили @Bean, возвращающий интерфейс CommandLineRunner. Spring Boot выполняет этот компонент после завершения связывания всех компонентов Spring воедино, но до запуска приложения; это удобный способ выполнения кода (например, инициализации) до запуска приложения.

Flux<T>, асинхронная последовательность [0|N] элементов

Flux представляет собой разновидность `Publisher<T>`, являющуюся асинхронной последовательностью от 0 до N выдаваемых элементов, которая может (не обязательно) завершаться с помощью сигнала `onComplete` или `onError` (рис. 6.3).

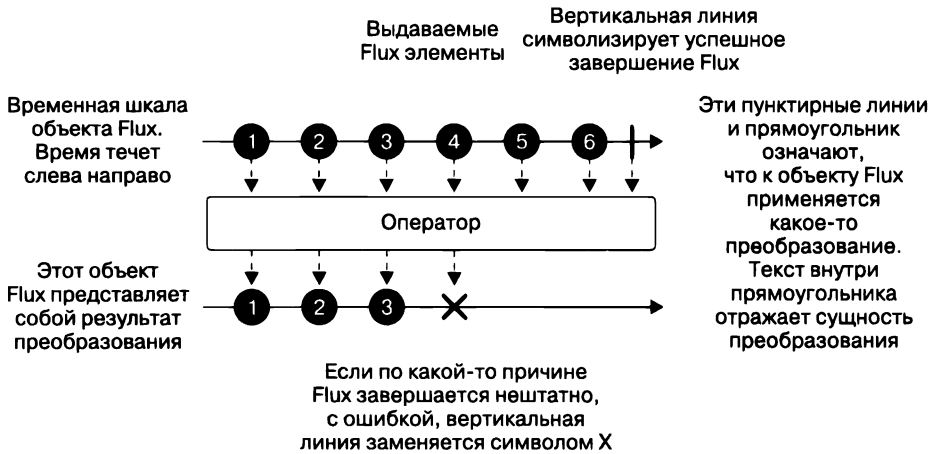


Рис. 6.3. Flux [0 | N] (изображение взято из документации http://projectreactor.io/)

Создайте класс `FluxExample` (листинг 6.3).

Листинг 6.3. com.apress.reactor.example.FluxExample.java

```
package com.apress.reactor.example;

import com.apress.reactor.example.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import reactor.core.publisher.EmitterProcessor;
import reactor.core.publisher.Mono;
import reactor.core.scheduler.Schedulers;

import java.util.List;

@Configuration
public class FluxExample {

    static private Logger LOG = LoggerFactory.getLogger(FluxExample.class);
    @Bean
    public CommandLineRunner runFluxExample(){
       return args -> {
       EmitterProcessor<ToDo> stream =
                        EmitterProcessor.create();
       Flux<List<ToDo>> promise = stream
                    .filter( s -> s.isCompleted())
                    .doOnNext(s -> LOG.info("FLUX >>> ToDo: {}",
                    s.getDescription()))
```

```
                .collectList()
                .subscribeOn(Schedulers.single());

        stream.onNext(new ToDo("Read a Book",true));
        stream.onNext(new ToDo("Listen Classical Music",true));
        stream.onNext(new ToDo("Workout in the Mornings"));
        stream.onNext(new ToDo("Organize my room", true));
        stream.onNext(new ToDo("Go to the Car Wash", true));
        stream.onNext(new ToDo("SP1 2018 is coming" , true));

            stream.onComplete();

            promise.block();
        };
    }
}
```

В листинге 6.3 приведен класс `FluxExample`; внимательно его изучим.

- `EmitterProcessor`. Как вы помните, узел-обработчик — разновидность издателя; в данном случае мы используем синхронный узел-обработчик, способный «проталкивать» данные как посредством действий пользователя, так и путем подписки на расположенный ранее по конвейеру издатель и синхронного «вытягивания» из него данных. Этот узел-обработчик создает Flux из экземпляров `ToDo`; он предоставляет реализацию основанного на `RingBuffer` *передающего сообщения* узла-обработчика, реализующего обмен сообщениями по типу «публикация/подписка» с синхронными циклами «вытягивания» данных. Если вы хотите применять асинхронный узел-обработчик — можете воспользоваться классом `WorkQueueProcessor` или `TopicProcessor`.
- `filter`. Напомню, что к API Flux и Mono можно применять операции; в данном случае мы используем фильтр с помощью вычисления предиката и выдачи значения в случае успеха (то есть если `ToDo` завершен).
- `doOnNext`. Срабатывает, когда Flux выдает значение.
- `collectList`. Собирает все выдаваемые Flux элементы в список, выдаваемый итоговым Mono при завершении этой последовательности.
- `subscribeOn`. Подписывается на данный Flux на основе потока-исполнителя `Scheduler`.
- `onNext`. Выдает во Flux начальное значение.
- `onComplete`. Завершает предыдущую часть конвейера.
- `block`. Подписывается на тип `Flux` и осуществляет блокировку до момента получения следующего сигнала или истечения времени ожидания.

Теперь можете запустить код. Вы должны увидеть примерно следующий вывод:

```
INFO 61 - [single-1] c.a.r.e.FluxExample : FLUX >>> ToDo: Read a Book
INFO 61 - [single-1] c.a.r.e.FluxExample : FLUX >>> ToDo: Listen Classical
Music
INFO 61 - [single-1] c.a.r.e.FluxExample : FLUX >>> ToDo: Organize my room
INFO 61 - [single-1] c.a.r.e.FluxExample : FLUX >>> ToDo: Go to the Car Wash
INFO 61 - [single-1] c.a.r.e.FluxExample : FLUX >>> ToDo: SP1 2018 is coming
```

Опять же это простой пример для приложения ToDo. Но представьте себе, что к вашему приложению обращаются миллионы пользователей для занесения запланированных дел в свои учетные записи и необходимо отслеживать их все — подобно ленте Twitter, — причем нежелательно блокировать каких-либо пользователей. Без реактивности не обойтись!

WebFlux

На протяжении долгого времени основным способом создания веб-приложений с помощью фреймворка Spring оставался Spring MVC. Теперь на сцену выходит новый игрок — реактивный стек, фреймворк Spring WebFlux! Spring WebFlux — полностью асинхронный и неблокирующий фреймворк, основанный на реализациях реактивных потоков данных из проекта Reactor.

Одна из главных возможностей Spring WebFlux — две модели программирования.

- *Снабженные аннотациями контроллеры.* В едином стиле с Spring MVC, на основе таких же аннотаций из модуля `spring-web`; это значит, что можно использовать уже знакомые вам аннотации (`@*Controller`, `@*Mapping`, `@RequestBody` и т. д.), но с полной поддержкой реактивности от Reactor и RxJava.

```
@RestController
public class ToDoController {
      @GetMapping("/todo/{id}")
      public Mono<ToDo> getToDo(@PathVariable Long id) {
            // ...
      }
      @GetMapping("/todo")
      public Flux<ToDo> getToDos() {
            // ...
      }
}
```

- *Функциональные конечные точки.* Модель функционального программирования с использованием вызовов на основе лямбда-выражений. Необходимо только указать конечные точки маршрутов, объявив компонент `RouterFunction` и обработчик конечной точки, возвращающий ответ типа `Mono` или `Flux`.

```
@Configuration
public class ToDoRoutingConfiguration {
      @Bean
      public RouterFunction<ServerResponse>
                  monoRouterFunction(ToDoHandler toDoHandler) {
            return
      route(GET("/todo/{id}")
        .and(accept(APPLICATION_JSON)),toDoHandler::getToDo)
        .andRoute(GET("/todo")
        .and(accept(APPLICATION_JSON)), toDoHandler::getToDos);
      }
}
@Component
public class ToDoHandler {
  public Mono<ServerResponse> getToDo(ServerRequest request){
            // ...
      }
  public Mono<ServerResponse> getToDos(ServerRequest request){
            // ...
      }
}
```

Подход к *фильтрам, обработчикам исключений, CORS*[1], *технологии представлений* и *веб-безопасности* не отличается от такового в Spring MVC. В этом и состоит преимущество фреймворка Spring — согласованная экосистема, независимо от появляющихся новых технологий.

WebClient

Модуль WebClient — реактивный неблокирующий клиент для HTTP-запросов с API в функциональном стиле и поддержкой реактивных потоков данных. Отличительные признаки интерфейса `WebClient`:

- функциональный API со всеми преимуществами программирования с использованием лямбда-выражений;
- неблокирующий и реактивный;

[1] См. https://ru.wikipedia.org/wiki/Cross-origin_resource_sharing.

- поддерживает высокую степень параллелизма при небольшом расходе аппаратных ресурсов;
- поддерживает потоковую передачу данных на сервер/с сервера;
- поддерживает как синхронный, так и асинхронный обмен сообщениями.

Использовать `WebClient` очень легко. У него есть методы `retrieve` и `exchange` для получения тела ответа и его декодирования.

```
WebClient client = WebClient.create("http://my-to-dos.com");
// [0|1] объект ToDo
Mono<ToDo> result = client
      .get()
      .uri("/todo/{id}", id)
      .accept(MediaType.APPLICATION_JSON)
      .retrieve()
      .bodyToMono(ToDo.class);

// [0|N] объектов ToDo
Flux<ToDo> result = client
     .get()
     .uri("/todo").accept(MediaType.TEXT_EVENT_STREAM)
     .retrieve()
     .bodyToFlux(ToDo.class);
```

Позднее мы создадим небольшой пример использования этого клиента, но если вам интересно узнать о нем больше, загляните на страницу https://docs.spring.io/spring/docs/current/spring-framework-reference/web-reactive.html#webflux-client.

WebFlux и автоконфигурация Spring Boot

С помощью Spring Boot использовать WebFlux еще проще, поскольку Spring Boot обеспечивает *автоконфигурацию* в виде настройки необходимых кодировщиков-декодировщиков для экземпляров `HttpMessageReader` и `HttpMessageWriter`. Он поддерживает выдачу статических ресурсов, включая поддержку WebJars. В нем применяются новейшие технологии шаблонизаторов, поддерживающих WebFlux, например FreeMarker (https://freemarker.apache.org/), Thymeleaf (www.thymeleaf.org) и Mustache (https://mustache.github.io/). По умолчанию автоконфигурация настраивает в качестве основного контейнера Netty (https://netty.io).

При необходимости переопределить автоконфигурацию WebFlux по умолчанию можно добавить свой собственный класс `@Configuration` типа `WebFluxConfigurer`.

ВАЖНОЕ ПРИМЕЧАНИЕ

Для полного контроля над автоконфигурацией WebFlux необходимо добавить свой собственный класс @Configuration, снабженный аннотацией @EnableWebFlux.

Использование WebFlux со Spring Boot

Для использования WebFlux со Spring Boot необходимо добавить зависимость `spring-boot-starter-webflux` в файл `pom.xml` или `build.gradle`.

ВАЖНОЕ ПРИМЕЧАНИЕ

Можно использовать и spring-boot-starter-web, и spring-boot-starter-webflux, но для автоматической настройки Spring Boot нужен Spring MVC. Для задействования всех возможностей WebFlux при этом нужно будет применить SpringApplication.setWebApplicationType(WebApplicationType.REACTIVE).

Создание приложения ToDo с использованием WebFlux

Начнем с создания приложения ToDo с использованием модуля WebFlux. И задействуем при этом новые реактивные API Flux и Mono.

Откройте браузер и перейдите в Spring Initializr. Внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-webflux`.
- Name (Название): `todo-webflux`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Lombok`, `Reactive Web`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами IDE (рис. 6.4).

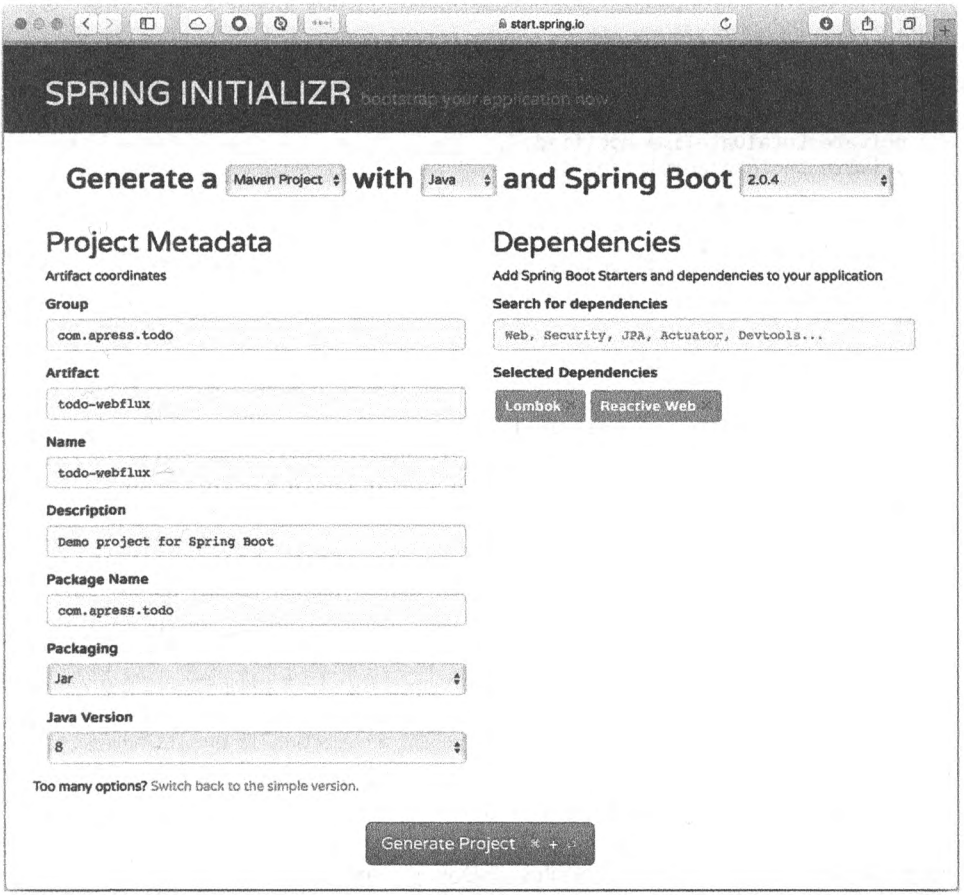


Рис. 6.4. Spring Initializr (https://start.spring.io)

На этот раз мы воспользуемся зависимостью Reactive Web; в данном случае `spring-boot-starter-webflux`. Создадим класс предметной области `ToDo` аналогично прочим проектам (листинг 6.4).

Листинг 6.4. com.apress.todo.domain.ToDo.java

```
package com.apress.todo.domain;

import lombok.Data;

import java.time.LocalDateTime;
import java.util.UUID;
```

```
@Data
public class ToDo {
    private String id;
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
        this.id = UUID.randomUUID().toString();
        this.created = LocalDateTime.now();
        this.modified = LocalDateTime.now();
    }

    public ToDo(String description){
        this();
        this.description = description;
    }

    public ToDo(String description, boolean completed){
        this();
        this.description = description;
        this.completed = completed;
    }
}
```

В листинге 6.4 приведен класс `ToDo`; как видите, ничего нового, за исключением инициализации в конструкторе по умолчанию.

Теперь создадим класс `ToDoRepository` для хранения списка запланированных дел в оперативной памяти (листинг 6.5).

Листинг 6.5. com.apress.todo.repository.ToDoRepository.java

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import java.util.Arrays;

public class ToDoRepository {

    private Flux<ToDo> toDoFlux =
       Flux.fromIterable(Arrays.asList(
            new ToDo("Do homework"),
            new ToDo("Workout in the mornings", true),
            new ToDo("Make dinner tonight"),
            new ToDo("Clean the studio", true)));

    public Mono<ToDo> findById(String id){
       return Mono
```

```
            .from(
            toDoFlux.filter( todo -> todo.getId().equals(id)));
    }

    public Flux<ToDo> findAll(){
        return toDoFlux;
    }
}
```

В листинге 6.5 приведен класс `ToDoRepository`, в котором экземпляр `toDoFlux` обрабатывает `Flux<ToDo>`. Обратите внимание на метод `findById`, возвращающий `Mono<ToDo>` из `Flux`.

Использование снабженных аннотациями контроллеров

Продолжим наш пример созданием кое-чего уже знакомого вам (аналогичного Spring MVC): класса `ToDoController`, отвечающего за обработку типов `Flux` и `Mono` (листинг 6.6).

Листинг 6.6. com.apress.todo.controller.ToDoController.java

```
package com.apress.todo.controller;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

@RestController
public class ToDoController {

    private ToDoRepository repository;

    public ToDoController(ToDoRepository repository){
        this.repository = repository;
    }

    @GetMapping("/todo/{id}")
    public Mono<ToDo> getToDo(@PathVariable String id){
        return this.repository.findById(id);
    }

    @GetMapping("/todo")
    public Flux<ToDo> getToDos(){
        return this.repository.findAll();
    }
}
```

Как можно видеть из кода, возвращается `Mono<ToDo>` или `Flux<ToDo>`, в отличие от Spring MVC. Помните, что производимые нами вызовы — асинхронные и неблокирующие.

Можете запустить приложение и перейти в браузере по адресу http://localhost:8080/todo, чтобы посмотреть на результаты — ответ в формате JSON, состоящий из списка запланированных дел; или можете выполнить следующую команду в терминале и получить тот же вывод:

```
$ curl http://localhost:8080/todo
[
    {
        "completed": false,
        "created": "2018-08-14T20:46:05.542",
        "description": "Do homework",
        "id": "5520e646-47aa-4be6-802a-ef6df500d6fb",
        "modified": "2018-08-14T20:46:05.542"
    },
    {
        "completed": true,
        "created": "2018-08-14T20:46:05.542",
        "description": "Workout in the mornings",
        "id": "3fe07f8d-64b0-4a39-ab1b-658bde4815d7",
        "modified": "2018-08-14T20:46:05.542"
    },
      ...
]
```

Обратите внимание в журналах консоли, что используется контейнер Netty в соответствии с настройками по умолчанию из автоконфигурации Spring Boot.

Использование функциональных конечных точек

Как вы помните, Spring WebFlux использует функциональное программирование для создания реактивных веб-приложений. Создадим класс `ToDoRouter` для объявления функций маршрутизации (листинг 6.7).

Листинг 6.7. com.apress.todo.reactive.ToDoRouter.java

```
package com.apress.todo.reactive;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerResponse;

import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.
  RequestPredicates.GET;
import static org.springframework.web.reactive.function.server.
  RequestPredicates.accept;
import static org.springframework.web.reactive.function.server.
  RouterFunctions.route;

@Configuration
public class ToDoRouter {
    @Bean
    public RouterFunction<ServerResponse> monoRouterFunction(ToDoHandler
        toDoHandler) {
        return route(GET("/todo/{id}").and(accept(APPLICATION_JSON)),
            toDoHandler::getToDo)
                .andRoute(GET("/todo").and(accept(APPLICATION_JSON)),
                    toDoHandler::getToDos);
    }
}
```

В листинге 6.7 показаны используемые маршруты (конечные точки) и их обработчик. Spring WebFlux предоставляет весьма удобный текучий API для построения произвольных маршрутов.

Создадим класс `ToDoHandler`, содержащий логику обслуживания запросов (листинг 6.8).

Листинг 6.8. com.apress.todo.reactive.ToDoHandler.java

```
package com.apress.todo.reactive;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.BodyInserters.
fromObject;

@Component
public class ToDoHandler {
```

```
    private ToDoRepository repository;

    public ToDoHandler(ToDoRepository repository){
        this.repository = repository;
    }

    public Mono<ServerResponse> getToDo(ServerRequest request) {
        String toDoId = request.pathVariable("id");
        Mono<ServerResponse> notFound =
            ServerResponse.notFound().build();
        Mono<ToDo> toDo = this.repository.findById(toDoId);
        return toDo
                .flatMap(t ->
                         ServerResponse
                        .ok()
                        .contentType(APPLICATION_JSON)
                        .body(fromObject(t)))
                .switchIfEmpty(notFound);
    }

    public Mono<ServerResponse> getToDos(
                                          ServerRequest request) {
        Flux<ToDo> toDos = this.repository.findAll();
        return ServerResponse
                .ok()
                .contentType(APPLICATION_JSON)
                .body(toDos, ToDo.class);
    }
}
```

В листинге 6.8 показан обработчик. Обсудим его подробнее.

`Mono<ServerResponse>`. Оба метода возвращают этот тип ответа, причем используется интерфейс `ServerResponse` с текучим API (Fluent API) `BodyBuilder` для добавления тела в ответ. Интерфейс `BodyBuilder` содержит удобные для формирования ответа (например, включения в него статуса с помощью метода `ok`) методы. Можно также добавлять заголовки с помощью метода `headers` и т. д.

Прежде чем запустить приложение, обязательно удалите класс `ToDoController` из контейнера Spring. Сделать это можно, закомментировав аннотацию `@RestController`.

При запуске приложения вы получите те же результаты, что и ранее. Конечно, это очень простой пример, ведь наше приложение все делает в оперативной памяти, правда? Что ж, добавим в него реактивное хранение данных!

Реактивные данные

Команда Spring Data создала реактивные репозитории с динамическими API, реализованные с помощью RxJava и Project Reactor. В этой абстракции описано несколько типов-оберток (wrapper types). Spring Data «за кулисами» преобразует реактивные типы-обертки, благодаря чему вы можете по-прежнему использовать свой любимый набор библиотек. Это сильно упрощает жизнь разработчика.

- `ReactiveCrudRepository`.
- `ReactiveSortingRepository`.
- `RxJava2CrudRepository`.
- `RxJava2SortingRepository`.

Spring Data предоставляет различные реактивные модули: MongoDB, Redis и реактивные потоки данных Cassandra, благодаря чему можно использовать всю гибкость и все интересные возможности реактивных потоков данных.

Реактивные потоки данных MongoDB

Модуль MongoDB Spring Data создан на основе реактивных потоков данных MongoDB и обеспечивает максимальное взаимодействие реактивных потоков данных. Он предоставляет возможность использования типов `Flux` и `Mono` с помощью вспомогательного интерфейса `ReactiveMongoOperations`.

Для использования реактивных потоков данных MongoDB необходимо включить зависимость `spring-boot-starter-data-mongodb-reactive` в файл `pom.xml` или `build.gradle`. Реактивные потоки данных MongoDB также предоставляют специализированный интерфейс для объявления репозитория — `ReactiveMongoRepository<T,ID>`. Следуя тому же паттерну репозитория, можно объявить и собственный *поименованный метод запроса*, возвращающий тип `Flux` или `Mono`.

Приложение ToDo с реактивными данными

Завершим наше приложение ToDo добавлением реактивных данных с помощью MongoDB, по-прежнему используя WebFlux для всех запросов и ответов в стиле функционального программирования.

Откройте свой любимый браузер и зайдите на сайт Spring Initializr (https://start.spring.io); внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-reactive-data`.
- Name (Название): `todo-reactive-data`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Lombok`, `Reactive Web`, `Reactive MongoDB`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами IDE (рис. 6.5).

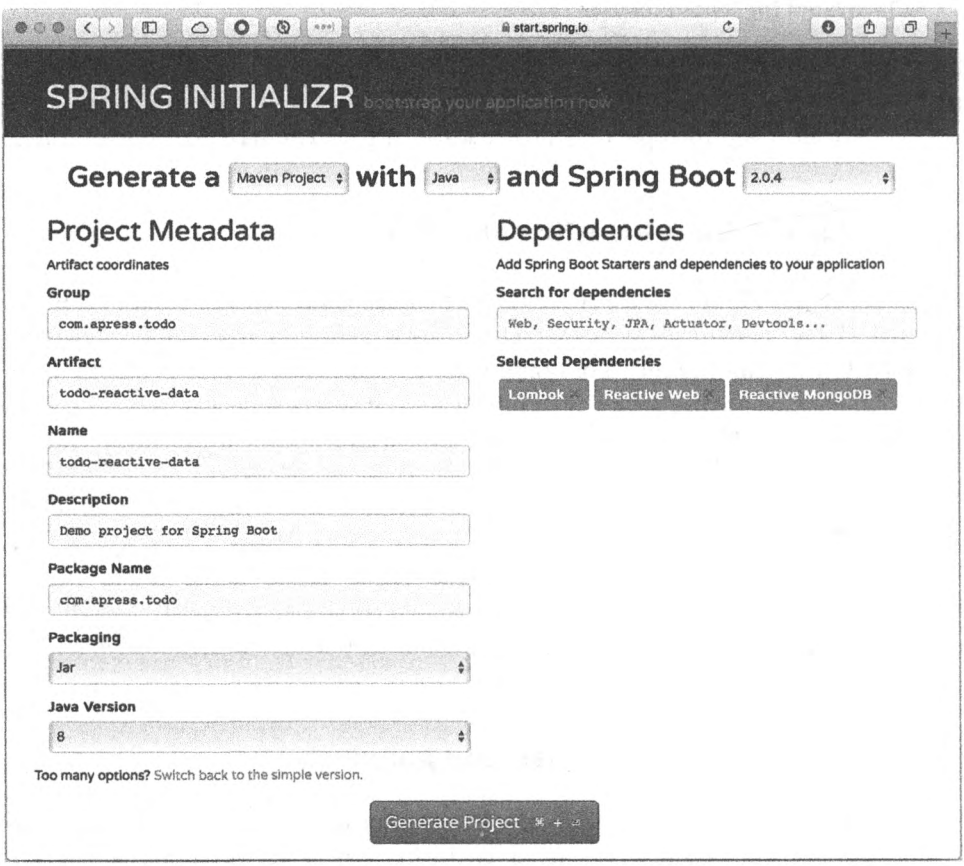


Рис. 6.5. Spring Initializr (https://start.spring.io)

Как вы видите, здесь появились зависимости Reactive Web и Reactive MongoDB. Благодаря применению MongoDB необходимости в сервере нет. Мы воспользуемся встроенным сервером Mongo; обычно его применяют для тестирования, но для нашего приложения этого достаточно.

Начнем с добавления зависимости для встроенного сервера Mongo. Если вы используете Maven, откройте файл `pom.xml` и добавьте в него следующую зависимость:

```
<dependency>
     <groupId>de.flapdoodle.embed</groupId>
     <artifactId>de.flapdoodle.embed.mongo</artifactId>
     <scope>runtime</scope>
  </dependency>
```

Если вы используете Gradle, добавьте следующую зависимость в файл `build.gradle`:

```
runtime('de.flapdoodle.embed:de.flapdoodle.embed.mongo')
```

Далее создадим класс предметной области `ToDo` (листинг 6.9).

Листинг 6.9. com.apress.todo.domain.ToDo.java

```
package com.apress.todo.domain;

import lombok.Data;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import java.time.LocalDateTime;
import java.util.UUID;

@Document
@Data
public class ToDo {

    @Id
    private String id;
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){
        this.id = UUID.randomUUID().toString();
        this.created = LocalDateTime.now();
        this.modified = LocalDateTime.now();
    }
```

```
    public ToDo(String description){
        this();
        this.description = description;
    }

    public ToDo(String description, boolean completed){
        this();
        this.description = description;
        this.completed = completed;
    }
}
```

Этот класс снабжен аннотациями `@Document` и `@Id`, указывающими, что он является классом постоянного хранения данных для MongoDB.

Создадим интерфейс ToDoRepository (листинг 6.10).

Листинг 6.10. com.apress.todo.repository.ToDoRepository.java

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.mongodb.repository.ReactiveMongoRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface ToDoRepository extends
            ReactiveMongoRepository<ToDo, String> {
}
```

Этот интерфейс расширяет интерфейс `ReactiveMongoRepository<T,ID>` и обеспечивает уже знакомую вам функциональность `Repository`, но теперь возвращает типы `Flux` и `Mono`. Все то же, что и в предыдущих главах. Здесь вы описываете пользовательские поименованные запросы, возвращающие реактивные типы данных.

Для этого приложения ToDo мы будем использовать реактивное программирование. Создадим функцию маршрутизации и обработчик в классе `ToDoRouter` (листинг 6.11).

Листинг 6.11. com.apress.todo.reactive.ToDoRouter.java

```
package com.apress.todo.reactive;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.web.reactive.function.server.RouterFunction;
import org.springframework.web.reactive.function.server.ServerResponse;

import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.server.
RequestPredicates.*;
import static org.springframework.web.reactive.function.server.
RouterFunctions.route;

@Configuration
public class ToDoRouter {

    @Bean
    public RouterFunction<ServerResponse>
            monoRouterFunction(ToDoHandler toDoHandler) {
        return
        route(GET("/todo/{id}").and(accept(APPLICATION_JSON)),
            toDoHandler::getToDo)
        .andRoute(GET("/todo").and(accept(APPLICATION_JSON)),
            toDoHandler::getToDos)
        .andRoute(POST("/todo").and(accept(APPLICATION_JSON)),
            toDoHandler::newToDo);
    }
}
```

В листинге 6.11 приведены объявления конечных точек. Здесь появился новый HTTP-метод, `POST`. Теперь создадим обработчик (листинг 6.12).

Листинг 6.12. com.apress.todo.reactive.ToDoHandler.java

```
package com.apress.todo.reactive;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.springframework.stereotype.Component;
import org.springframework.web.reactive.function.server.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;

import static org.springframework.http.MediaType.APPLICATION_JSON;
import static org.springframework.web.reactive.function.BodyInserters.
  fromObject;
import static org.springframework.web.reactive.function.BodyInserters.
  fromPublisher;
@Component
public class ToDoHandler {
```

```
    private ToDoRepository repository;

    public ToDoHandler(ToDoRepository repository){
        this.repository = repository;
    }

    public Mono<ServerResponse> getToDo(ServerRequest request) {
        return findById(request.pathVariable("id"));
    }

    public Mono<ServerResponse> getToDos(ServerRequest request) {
        Flux<ToDo> toDos = this.repository.findAll();
        return ServerResponse
                .ok()
                .contentType(APPLICATION_JSON)
                .body(toDos, ToDo.class);
    }

    public Mono<ServerResponse> newToDo(ServerRequest request) {
        Mono<ToDo> toDo = request.bodyToMono(ToDo.class);
        return ServerResponse
                .ok()
                .contentType(APPLICATION_JSON)
                .body(fromPublisher(toDo.flatMap(this::save),ToDo.class));
    }

    private Mono<ServerResponse> findById(String id){
        Mono<ToDo> toDo = this.repository.findById(id);
        Mono<ServerResponse> notFound = ServerResponse.notFound().build();
        return toDo
                .flatMap(t -> ServerResponse
                        .ok()
                        .contentType(APPLICATION_JSON)
                        .body(fromObject(t)))
                .switchIfEmpty(notFound);
    }

    private Mono<ToDo> save(ToDo toDo) {
        return Mono.fromSupplier(
                () -> {
                    repository
                            .save(toDo)
                            .subscribe();
                    return toDo;
                });
    }
}
```

В листинге 6.12 приведен обработчик, отвечающий за ответ на обращение к конечным точкам. Все методы возвращают `Mono<ServerResponse>`. Взгляните на операторы `Mono`, обеспечивающие сохранение данных в сервер MongoDB, и на то, как `BodyBuilder` формирует ответ.

Создайте конфигурацию с настройками соединения со встроенным сервером MongoDB. Создайте класс `ToDoConfig` (листинг 6.13).

Листинг 6.13. com.apress.todo.config.ToDoConfig.java

```
package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import com.mongodb.reactivestreams.client.MongoClient;
import com.mongodb.reactivestreams.client.MongoClients;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.DependsOn;
import org.springframework.core.annotation.Order;
import org.springframework.core.env.Environment;
import org.springframework.data.mongodb.config.
  AbstractReactiveMongoConfiguration;
import org.springframework.data.mongodb.repository.config.
  EnableReactiveMongoRepositories;

import java.util.Arrays;
import java.util.Optional;
import java.util.stream.Collectors;

@Configuration
@EnableReactiveMongoRepositories(basePackages = "com.apress.todo.repository")
public class ToDoConfig extends AbstractReactiveMongoConfiguration {

    private final Environment environment;

    public ToDoConfig(Environment environment){
        this.environment = environment;
    }

    @Override
    protected String getDatabaseName() {
        return "todos";
    }
```

```
    @Override
    @Bean
    @DependsOn("embeddedMongoServer")
    public MongoClient reactiveMongoClient() {
         int port = environment.getProperty("local.mongo.port",
             Integer.class);
         return MongoClients.create(String.format("mongodb://localhost:%d",
             port));
    }

    @Bean
     public CommandLineRunner insertAndView(ToDoRepository repository,
        ApplicationContext context){
        return args -> {
            repository.save(new ToDo("Do homework")).subscribe();
            repository.save(new ToDo("Workout in the mornings", true)).
                 subscribe();
            repository.save(new ToDo("Make dinner tonight")).subscribe();
            repository.save(new ToDo("Clean the studio", true)).subscribe();
            repository.findAll().subscribe(System.out::println);
        };
    }
}
```

В листинге 6.13 показана конфигурация, необходимая для использования реактивных потоков данных встроенного сервера MongoDB. Внимательно проанализируем этот класс.

- Аннотация `@EnableReactiveMongoRepositories`. Эта аннотация необходима для настройки всей необходимой для API реактивных потоков данных инфраструктуры. Важно также указать в ней местоположение репозиториев (не обязательно, если репозитории входят в состав пакета).
- Класс `AbstractReactiveMongoConfiguration`. Для настройки встроенного сервера MongoDB необходимо расширить этот абстрактный класс, реализовав методы `reactiveMongoClient` и `getDatabaseName`. Метод `reactiveMongoClient` создает экземпляр `MongoClient`, подключающийся к порту, на котором работает встроенный сервер MongoDB (задается с помощью свойства среды `local.mongo.port`).
- Аннотация `@DependsOn`. Эта аннотация представляет собой вспомогательный интерфейс, предназначенный для создания `reactiveMongoClient` после компонента `embeddedMongoServer`.

Этот класс также отвечает за выполнение кода, в котором запланированные дела сохраняются в MongoDB.

Теперь мы готовы запустить наше приложение ToDo. После его запуска вы можете перейти в браузер или выполнить следующую команду.

```
$ curl http://localhost:8080/todo
[
    {
        "completed": false,
        "created": "2018-08-14T20:46:05.542",
        "description": "Do homework",
        "id": "5520e646-47aa-4be6-802a-ef6df500d6fb",
        "modified": "2018-08-14T20:46:05.542"
    },
    {
        "completed": true,
        "created": "2018-08-14T20:46:05.542",
        "description": "Workout in the mornings",
        "id": "3fe07f8d-64b0-4a39-ab1b-658bde4815d7",
        "modified": "2018-08-14T20:46:05.542"
    },
      ...
]
```

Можете добавить новое запланированное дело, выполнив следующее:

```
$ curl -i -X POST -H "Content-Type: application/json" -d '{
"description":"Read a book"}' http://localhost:8080/todo
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 164

{
  "id":"a3133b8d-1d8b-4b2e-b7d9-48a73999a104",
  "description":"Read a book",
  "created":"2018-08-14T22:51:19.734",
  "modified":"2018-08-14T22:51:19.734",
  "completed":false
}
```

Поздравляем! Теперь вы знаете, как создать реактивное, асинхронное и неблокирующее приложение с помощью возможностей Spring WebFlux и проекта Reactor!

ПРИМЕЧАНИЕ

Весь исходный код доступен на веб-сайте Apress. Найти самую свежую его версию можно также в репозитории GitHub: https://github.com/Apress/pro-spring-boot-2.

Резюме

В этой главе обсуждалось использование WebFlux — нового компонента фреймворка Spring. Теперь вы знаете, что WebFlux — неблокирующий, асинхронный и реактивный фреймворк, реализующий реактивные потоки данных.

Вы также узнали, что WebFlux дает возможность использовать две модели программирования со Spring Boot: снабженные аннотациями контроллеры и функциональные конечные точки, в которых можно описывать ответы на основе типов Flux и Mono. Вы узнали, что WebFlux также предоставляет для использования этих новых реактивных API интерфейс `WebClient`.

Вы узнали, что Spring Boot предоставляет автоконфигурацию для реактивного стека посредством зависимости `spring-boot-starter-webflux`, причем в качестве контейнера веб-сервера по умолчанию используется Netty.

Следующая глава посвящена тестированию приложений. В ней мы продемонстрируем, насколько Spring Boot упрощает разработку через тестирование.

7 Тестирование

Тестирование программного обеспечения — процесс запуска программы или системы, цель которого — найти (программные) ошибки и убедиться, что все программы или система в целом действительно работают.

Это одно из множества встречающихся в Интернете определений.

Многие компании прилагают всяческие усилия для поиска правильного и простого способа тестирования, создавая замечательные фреймворки — практика, называемая *разработкой через тестирование* (test-driven development, TDD).

TDD — процесс, основанный на повторении очень короткого цикла разработки; важнейшую роль в нем играет обратная связь, поскольку разработчик пишет код для прохождения сценария использования при минимальном объеме кода. А благодаря обратной связи можно производить рефакторинг кода до тех пор, пока тест не будет пройден и конечного пользователя не будет удовлетворять результат.

Фреймворк тестирования Spring

Одна из основных идей фреймворка Spring — поощрение создания разработчиками простых и слабо связанных классов, программирование интерфейсов, благодаря чему программное обеспечение становится более надежным и лучше расширяемым. Фреймворк Spring предоставляет инструменты, упрощающие модульное и интеграционное тестирование (на самом деле,

если вы действительно программируете интерфейсы, для тестирования функциональности приложений Spring вам не нужен). Другими словами, необходимо, чтобы приложение можно было протестировать с помощью систем тестирования JUnit или TestNG на основе объектов (создаваемых просто с помощью оператора `new` — без Spring или какого-либо другого контейнера).

В фреймворк Spring включено несколько пакетов, предназначенных для модульного или интеграционного тестирования приложений. Для модульного тестирования предназначено несколько имитационных объектов (`Environment`, `PropertySource`, `JNDI`, `Servlet`; реактивные утилиты тестирования `ServerHttpRequest` и `ServerHttpResponse`), с помощью которых можно производить изоляционное тестирование кода.

Одна из чаще всего используемых возможностей тестирования в фреймворке Spring — интеграционное тестирование. Основные его задачи:

- управление кэшированием контейнера Spring IoC в промежутке между выполнениями тестов;
- управление транзакциями;
- внедрение зависимостей экземпляров тестовых объектов;
- создание предназначенных специально для Spring базовых классов.

Фреймворк Spring обеспечивает легкий способ тестирования посредством интеграции в тесты контекста приложения (`ApplicationContext`). Модуль тестирования Spring предлагает несколько способов использования `ApplicationContext`, программным образом и с помощью аннотаций.

- Аннотация `BootstrapWith`. Аннотация уровня класса, предназначенная для конфигурации начальной загрузки фреймворка `TestContext` Spring.
- Аннотация `@ContextConfiguration`. Определяет метаданные уровня класса, задающие способ загрузки и настройки `ApplicationContext` для интеграционных тестов. Это совершенно необходимая для ваших классов аннотация, поскольку именно здесь `ApplicationContext` загружает все определения компонентов.
- Аннотация `@WebAppConfiguration`. Аннотация уровня класса, указывающая, что загружаемым для интеграционного теста контекстом приложения должно быть `WebApplicationContext`.

- Аннотация `@ActiveProfile`. Аннотация уровня класса, указывающая, какой из профилей определения компонентов должен быть активным при загрузке `ApplicationContext` для интеграционного теста.
- Аннотация `@TestPropertySource`. Аннотация уровня класса, предназначенная для задания местоположений файлов свойств и встраиваемых свойств, добавляемых в набор объектов `PropertySource` в `Environment` для загружаемого для интеграционного теста `ApplicationContext`.
- Аннотация `@DirtiesContext`. Указывает, что используемый `ApplicationContext` был «загрязнен» во время выполнения теста (например, модифицирован или поврежден путем изменения состояния компонента-одиночки) и должен быть закрыт.

Это далеко не все аннотации, существует множество других (`@TestExecutionListeners`, `@Commit`, `@Rollback`, `@BeforeTransaction`, `@AfterTransaction`, `@Sql`, `@SqlGroup`, `@SqlConfig`, `@Timed`, `@Repeat`, `@IfProfileValue` и т. д.).

Как вы видите, существует множество вариантов тестирования с помощью фреймворка Spring. Обычно всегда используется аннотация `@RunWith`, связывающая воедино все элементы фреймворка тестирования. Например:

```
@RunWith(SpringRunner.class)
@ContextConfiguration({"/app-config.xml", "/test-data-access-config.xml"})
@ActiveProfiles("dev")
@Transactional
public class ToDoRepositoryTests {

      @Test
      public void ToDoPersistenceTest(){
            //...
      }
}
```

Теперь взглянем, как следует использовать фреймворк тестирования Spring и какие возможности предоставляет Spring Boot.

Фреймворк тестирования Spring Boot

Spring Boot использует мощь фреймворка тестирования Spring, расширяя старые и добавляя новые аннотации и возможности, благодаря которым для разработчиков тестирование значительно упрощается.

Чтобы начать применять все возможности тестирования Spring Boot, необходимо лишь добавить в приложение зависимость `spring-boot-starter-test` с указанием области видимости `test` (`scope test`). В сервисе Spring Initializr эта зависимость уже добавлена.

Зависимость `spring-boot-starter-test` обеспечивает возможность использования нескольких фреймворков тестирования, очень хорошо согласующихся с возможностями тестирования Spring Boot: JUnit, AssertJ, Hamcrest, Mockito, JSONassert и JsonPath. Конечно, существуют и другие фреймворки тестирования, отлично работающие с модулем Spring Boot Test; просто соответствующие зависимости нужно указывать вручную.

Spring Boot предоставляет аннотацию `@SpringBootTest`, упрощающую тестирование приложений Spring. Обычно при тестировании приложения Spring необходимо добавить несколько аннотаций для тестирования конкретной возможности или функциональности приложения, но не в Spring Boot — хотя для тестирования все равно нужно указать аннотацию `@RunWith(SpringRunner.class)`; если этого не сделать, любые новые аннотации тестирования Spring Boot будут проигнорированы. У аннотации `@SpringBootTest` есть полезные для тестирования веб-приложений параметры, например `RANDOM_PORT` и `DEFINED_PORT`.

Следующий фрагмент кода представляет собой каркас теста Spring Boot.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MyTests {

      @Test
      public void exampleTest() {
            ...
      }
}
```

Тестирование конечных точек веб-приложения

Spring Boot предоставляет удобный способ тестирования конечных точек: имитационную среду под названием класс `MockMvc`:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureMockMvc
public class MockMvcToDoTests {
```

```
    @Autowired
    private MockMvc mvc;

    @Test
    public void toDoTest() throws Exception {
        this.mvc
        .perform(get("/todo"))
        .andExpect(status().isOk())
        .andExpect(content()
           .contentType(MediaType.APPLICATION_JSON_UTF8));
    }
}
```

Можно также воспользоваться классом `TestRestTemplate`.

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
public class ToDoSimpleRestTemplateTests {

    @Autowired
    private TestRestTemplate restTemplate;

    @Test
    public void toDoTest() {
        String body = this.restTemplate.getForObject("/todo", String.class);
        assertThat(body).contains("Read a Book");
    }
}
```

В этом коде показан тест, запускающий полноценный сервер и использующий экземпляр класса `TestRestTemplate` для обращения к конечной точке `/todo`. Здесь мы предполагаем, что возвращается объект `String` (это не лучший способ тестирования возврата JSON; не волнуйтесь, далее мы покажем, как использовать класс `TestRestTemplate` правильно).

Имитация компонент

Модуль тестирования Spring Boot предоставляет аннотацию `@MockBean`, описывающую имитационный объект *Mockito* для компонента в `ApplicationContext`. Другими словами, можно создать имитацию нового компонента Spring или заменить уже существующее определение, добавив эту аннотацию. Помните: все это происходит внутри `ApplicationContext`.

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ToDoSimpleMockBeanTests {
```

```
    @MockBean
    private ToDoRepository repository;

    @Test
    public void toDoTest() {
        given(this.repository.findById("my-id"))
            .Return(new ToDo("Read a Book"));
        assertThat(
            this.repository.findById("my-id").getDescription())
            .isEqualTo("Read a Book");
    }
}
```

Тестовые срезы Spring Boot

Одна из важнейших возможностей Spring Boot — выполнение тестов без необходимости в какой-то определенной инфраструктуре. Модуль тестирования Spring Boot включает так называемые *срезы* (slices), предназначенные для тестирования конкретных частей приложения без использования сервера или СУБД.

Аннотация @JsonTest

В модуле тестирования Spring Boot есть аннотация `@JsonTest`, упрощающая сериализацию/десериализацию JSON-объектов и проверяющая, все ли работает корректно. `@JsonTest` автоматически настраивает поддерживаемое средство JSON-отображения, в зависимости от найденной по пути к классам библиотеки: Jackson, GSON или JSONB.

```
@RunWith(SpringRunner.class)
@JsonTest
public class ToDoJsonTests {

    @Autowired
    private JacksonTester<ToDo> json;

    @Test
    public void toDoSerializeTest() throws Exception {
        ToDo toDo = new ToDo("Read a Book");

        assertThat(this.json.write(toDo))
        .isEqualToJson("todo.json");
```

```
        assertThat(this.json.write(toDo))
        .hasJsonPathStringValue("@.description");

        assertThat(this.json.write(toDo))
        .extractingJsonPathStringValue("@.description")
        .isEqualTo("Read a Book");
    }

    @Test
    public void toDoDeserializeTest() throws Exception {
         String content = "{\"description\":\"Read a Book\",\"completed\":
             true }";
         assertThat(this.json.parse(content))
                .isEqualTo(new ToDo("Read a Book", true));
         assertThat(
           this.json.parseObject(content).getDescription())
         .isEqualTo("Read a Book");
    }
}
```

Для тестирования контроллеров без использования полноценного сервера можно воспользоваться предлагаемой Spring Boot аннотацией `@WebMvcTest`, которая автоматически настраивает инфраструктуру Spring MVC и ограничивает список просматриваемых компонентов следующими: `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter`, `Filter`, `WebMvcConfigurer` и `HandlerMethodArgumentResolver`; благодаря этому вы будете знать, так ли работают ваши контроллеры, как ожидалось.

Важно понимать, что помеченные как `@Component` компоненты не просматриваются при использовании этой аннотации, но при необходимости можно применить аннотацию `@MockBean`.

```
@RunWith(SpringRunner.class)
@WebMvcTest(ToDoController.class)
public class ToDoWebMvcTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private ToDoRepository toDoRepository;

    @Test
    public void toDoControllerTest() throws Exception {
        given(this.toDoRepository.findById("my-id"))
            .Return(new ToDo("Do Homework", true));
```

```
        this.mvc.perform(get("/todo/my-id").accept(MediaType.APPLICATION_
          JSON_UTF8))
            .andExpect(status().isOk()).andExpect(content().
              string("{\"id\":\"my-id\",\"description\":\"Do Homework\",
              \"completed\":true}"));
    }

}
```

Аннотация @WebFluxTest

Для реактивных контроллеров Spring Boot предоставляет аннотацию `@WebFluxTest`. Эта аннотация автоматически настраивает инфраструктуру модуля Spring WebFlux и ищет только `@Controller`, `@ControllerAdvice`, `@JsonComponent`, `Converter`, `GenericConverter` и `WebFluxConfigurer`.

Важно понимать, что помеченные как `@Component` компоненты не просматриваются при использовании этой аннотации, но при необходимости вы можете применить аннотацию `@MockBean`.

```
@RunWith(SpringRunner.class)
@WebFluxTest(ToDoFluxController.class)
public class ToDoWebFluxTest {

    @Autowired
    private WebTestClient webClient;

    @MockBean
    private ToDoRepository toDoRepository;

    @Test
    public void testExample() throws Exception {
         given(this.toDoRepository.findAll())
                 .Return(Arrays.asList(new ToDo("Read a Book"), new
                     ToDo("Buy Milk")));
         this.webClient.get().uri("/todo-flux").accept(MediaType.
             APPLICATION_JSON_UTF8)
                .exchange()
                .expectStatus().isOk()
                .expectBody(List.class);
    }
}
```

Аннотация @DataJpaTest

Для тестирования JPA-приложений модуль тестирования Spring Boot предоставляет аннотацию `@DataJpaTest`, производящую автоконфигурацию встроен-

ных баз данных, *размещаемых в оперативной памяти*. Она ищет `@Entity` и не загружает никаких компонентов `@Component`. Кроме того, она предоставляет вспомогательный класс `TestEntityManager`, ориентированный на тестирование, очень похожий на класс JPA `EntityManager`.

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class TodoDataJpaTests {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private ToDoRepository repository;

    @Test
    public void toDoDataTest() throws Exception {
        this.entityManager.persist(new ToDo("Read a Book"));
        Iterable<ToDo> toDos = this.repository.
             findByDescriptionContains("Read a Book");
        assertThat(toDos.iterator().next()).toString().contains("Read a Book");
    }

}
```

Учтите, что при тестировании с помощью `@DataJpaTest` используются встроенные СУБД в оперативной памяти. Для тестирования же с настоящей базой данных необходимо снабдить класс теста аннотацией `@AutoConfigureTestDatabase(replace=Replace.NONE)`.

```
@RunWith(SpringRunner.class)
@DataJpaTest
@AutoConfigureTestDatabase(replace=Replace.NONE)
public class TodoDataJpaTests {
      //...
}
```

Аннотация @JdbcTest

Эта аннотация очень похожа на `@DataJpaTest`; единственное отличие — она выполняет ориентированные исключительно на JDBC тесты. Она производит автоматическую настройку встроенной СУБД, размещаемой в оперативной памяти, и класса `JdbcTemplate`, пропуская при этом все снабженные аннотацией `@Component` классы.

```
@RunWith(SpringRunner.class)
@JdbcTest
```

```
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public class TodoJdbcTests {

    @Autowired
    private NamedParameterJdbcTemplate jdbcTemplate;

    private CommonRepository<ToDo> repository;

      @Test
      public void toDoJdbcTest() {
          ToDo toDo = new ToDo("Read a Book");

        this.repository = new ToDoRepository(jdbcTemplate);
        this.repository.save(toDo);

        ToDo result = this.repository.findById(toDo.getId());
        assertThat(result.getId()).isEqualTo(toDo.getId());
      }

}
```

Аннотация @DataMongoTest

Для тестирования приложений MongoDB модуль тестирования Spring Boot предоставляет аннотацию `@DataMongoTest`. Она производит автоматическую настройку встроенного размещаемого в оперативной памяти сервера Mongo, если он доступен; если же нет, необходимо добавить нужные свойства `spring.data.mongodb.*`. Она также производит настройку класса `MongoTemplate` и ищет аннотации `@Document`. Компоненты `@Component` пропускаются.

```
@RunWith(SpringRunner.class)
@DataMongoTest
public class TodoMongoTests {

    @Autowired
    private MongoTemplate mongoTemplate;

      @Test
      public void toDoMongoTest() {
        ToDo toDo = new ToDo("Read a Book");
        this.mongoTemplate.save(toDo);

        ToDo result = this.mongoTemplate.findById(toDo.getId(),ToDo.class);
        assertThat(result.getId()).isEqualTo(toDo.getId());
      }

}
```

При потребности во внешнем сервере MongoDB (невстроенном, размещаемом в оперативной памяти) добавьте в аннотацию `@DataMongoTest` параметр `excludeAutoConfiguration = EmbeddedMongoAutoConfiguration.class`.

```
@RunWith(SpringRunner.class)
@DataMongoTest(excludeAutoConfiguration = EmbeddedMongoAutoConfiguration.
class)
public class ToDoMongoTests {
      // ...
}
```

Аннотация @RestClientTest

Еще одна важная аннотация — `@RestClientTest`, предназначенная для тестирования REST-клиентов. Эта аннотация автоматически производит настройки для поддержки Jackson, GSON и JSONB, а также настраивает класс `RestTemplateBuilder` и добавляет поддержку `MockRestServiceServer`.

```
@RunWith(SpringRunner.class)
@RestClientTest(ToDoService.class)
public class ToDoRestClientTests {

    @Autowired
    private ToDoService service;

    @Autowired
    private MockRestServiceServer server;

    @Test
    public void toDoRestClientTest()
            throws Exception {
         String content = "{\"description\":\"Read a Book\",\"completed\":
             true }";
        this.server.expect(requestTo("/todo/my-id"))
         .andRespond(withSuccess(content,MediaType.APPLICATION_JSON_UTF8));
        ToDo result = this.service.findById("my-id");
        assertThat(result).isNotNull();
        assertThat(result.getDescription()).contains("Read a Book");
    }
}
```

Существует множество других доступных для использования срезов. Главное, запомнить: для тестирования необязательна полная инфраструктура со всеми запущенными серверами. Упростить тестирование приложений Spring Boot позволяют срезы.

Резюме

Из этой главы вы узнали о различных способах тестирования приложений Spring Boot. Несмотря на краткость этой главы, я продемонстрировал вам некоторые важные возможности, например работу со срезами.

В следующей главе мы обсудим вопрос безопасности и узнаем, как Spring Boot обеспечивает безопасность наших приложений.

8 Безопасность

В этой главе показано, как обеспечить безопасность веб-приложений с помощью Spring Boot. Мы расскажем вам обо всем, начиная от основ безопасности и до использования OAuth. За последнее десятилетие безопасность превратилась в важнейший фактор для веб-, традиционных и мобильных приложений. Но реализация безопасности — непростая задача, ведь приходится учитывать все: межсайтовое выполнение скриптов, авторизацию и аутентификацию, сеансы безопасной связи, идентификацию, шифрование и многое другое. Для реализации даже простейших средств безопасности в приложении необходимо сделать очень много.

Команда специалистов по безопасности Spring немало поработала, чтобы упростить для разработчиков обеспечение безопасности приложений, начиная от обеспечения безопасности методов сервисов и до целых веб-приложений. В центре архитектуры безопасности Spring лежат интерфейсы `AuthenticationProvider` и `AuthenticationManager`, а также специализированный `UserDetailsService`; кроме того, Spring обеспечивает интеграцию с поставщиками, такими как LDAP, Active Directory, Kerberos, PAM, OAuth и т. д., идентификационной информации. Мы рассмотрим некоторые из них в примерах из этой главы.

Spring Security

Spring Security — чрезвычайно широко настраиваемый и обладающий большими возможностями фреймворк для аутентификации и авторизации (или управления доступом); это модуль, используемый по умолчанию для

обеспечения безопасности приложений Spring. Вот некоторые из важнейших его возможностей.

- Интеграция API сервлетов.
- Интеграция со Spring Web MVC и WebFlux.
- Защита от таких атак, как фиксация сеанса, перехват кликов, межсайтовая подделка запросов (cross-site request forgery, CSRF), совместное использование ресурсов между разными источниками (cross-origin resource sharing, CORS) и т. д.
- Расширенная комплексная поддержка аутентификации и авторизации.
- Интеграция с технологиями базовой HTTP-аутентификации (HTTP Basic), дайджест-HTTP-аутентификации (HTTP Digest), X.509, LDAP, аутентификации на основе форм, OpenID, CAS, RMI, Kerberos, JAAS, Java EE и др.
- Интеграция со сторонними технологиями: AppFuse, DWR, Grails, Tapestry, JOSSO, AndroMDA, Roller и многими другими.

Spring Security стал де-факто способом обеспечения безопасности во многих проектах Java и Spring, поскольку интегрируется в них и настраивается под свои нужды с минимальными усилиями, позволяя создавать надежные и безопасные приложения.

Обеспечение безопасности с помощью Spring Boot

Для обеспечения безопасности приложений Spring Boot использует мощь фреймворка Spring Security. Для применения Spring Security необходимо добавить зависимость `spring-boot-starter-security`. Эта зависимость обеспечивает все основные JAR-файлы `spring-security` и автоматически настраивает стратегию определения используемого механизма аутентификации (`httpBasic` или `formLogin`). По умолчанию применяется `UserDetailsService` с одним пользователем. Имя этого пользователя и генерируемый случайным образом пароль выводятся в журнал с уровнем журналирования INFO при запуске приложения.

Другими словами, добавление зависимости `spring-boot-starter-security` уже обеспечивает безопасность приложения.

Приложение ToDo с базовым уровнем безопасности

Начнем создание приложения ToDo. Здесь используется тот же код, что и в проекте JPA REST, но мы обсудим соответствующий класс еще раз. Начните с нуля: перейдите в браузер и откройте сайт Spring Initializr (https://start.spring.io). Внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-simple-security`.
- Name (Название): `todo-simple-security`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Web`, `Security`, `Lombok`, `JPA`, `REST Repositories`, `H2`, `MySQL`, `Mustache`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в вашу любимую IDE (рис. 8.1).

В этом проекте теперь есть модуль Security и шаблонизатор Mustache. Очень скоро вы увидите, как его использовать.

Начнем с класса предметной области ToDo (листинг 8.1).

Листинг 8.1. com.apress.todo.domain.ToDo.java

```
package com.apress.todo.domain;

import lombok.Data;
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;

@Entity
@Data
public class ToDo {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
```

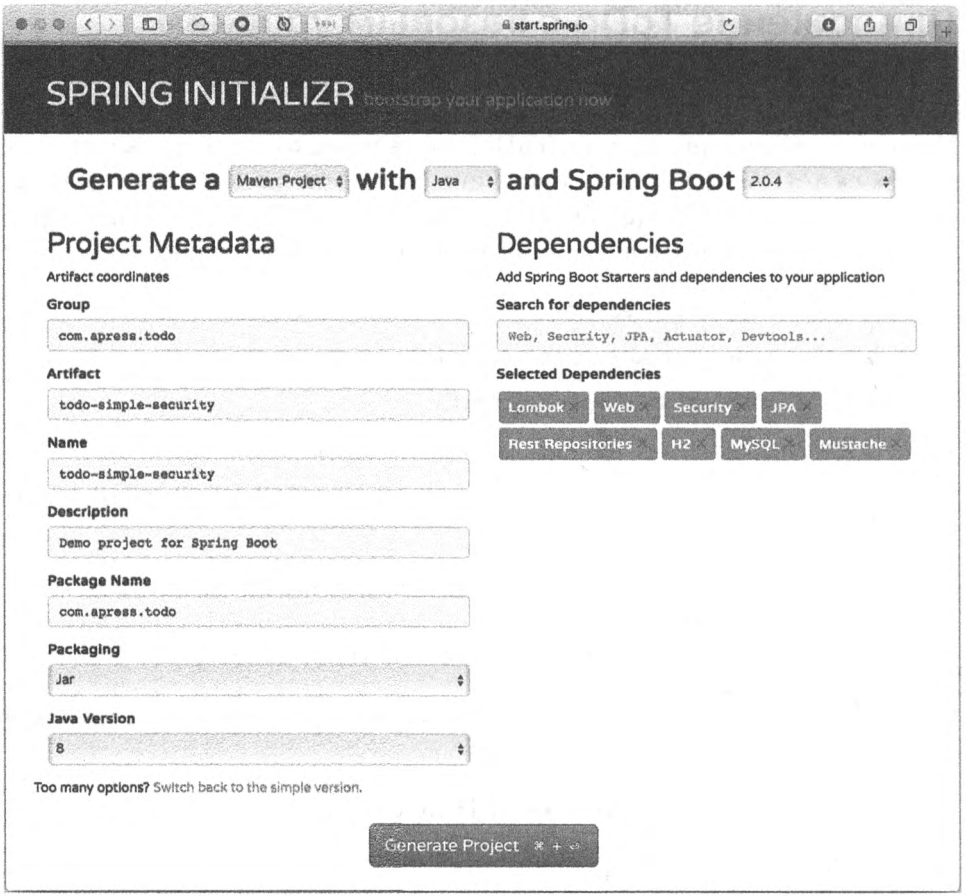


Рис. 8.1. Spring Initializr

```
private String id;
@NotNull
@NotBlank
private String description;

@Column(insertable = true, updatable = false)
private LocalDateTime created;
private LocalDateTime modified;
private boolean completed;

public ToDo(){}
public ToDo(String description){
    this.description = description;
}
```

```
    @PrePersist
    void onCreate() {
        this.setCreated(LocalDateTime.now());
        this.setModified(LocalDateTime.now());
    }
    @PreUpdate
    void onUpdate() {
        this.setModified(LocalDateTime.now());
    }
}
```

В листинге 8.1 приведен класс предметной области ToDo. Вам он уже знаком. Он снабжен аннотацией `@Entity` и использует `@Id` в качестве первичного ключа. Этот класс взят из проекта *todo-rest*.

Далее взглянем на интерфейс `ToDoRepository` (листинг 8.2).

Листинг 8.2. com.apress.todo.repository.ToDoRepository.java

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends CrudRepository<ToDo,String> {

}
```

В листинге 8.2 приведен интерфейс `ToDoRepository`, и, конечно, он тоже уже вам знаком. С помощью описания расширяющего `CrudRepository<T,ID>` интерфейса, включающего не только методы CRUD, но и Spring Data REST, мы создаем все API REST, необходимые для поддержки класса предметной области.

Взглянем на файл `application.properties` и посмотрим, что в нем нового (листинг 8.3).

Листинг 8.3. src/main/resources/application.properties

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true
# H2-Console: http://localhost:8080/h2-console

# jdbc:h2:mem:testdb
spring.h2.console.enabled=true

# REST API
spring.data.rest.base-path=/api
```

В листинге 8.3 приведен файл `application.properties`. Некоторые из этих свойств уже вам знакомы, за исключением последнего, правда? Свойство `spring.data.rest.base-path` указывает RestController (из конфигурации Spring Data REST) использовать `/api` в качестве корневой точки для всех конечных точек API REST. Так что для получения списка запланированных дел необходимо обратиться к конечной точке http://localhost:8080/api/toDos.

Прежде чем запустить приложение, добавим в него конечные точки с помощью SQL-сценария. Создайте файл `main/resources/data.sql`, содержащий следующие SQL-операторы:

```
insert into to_do (id,description,created,modified,completed)
values ('8a8080a365481fb00165481fbca90000', 'Read a Book','2018-08-17
07:42:44.136','2018-08-17 07:42:44.137',true);
insert into to_do (id,description,created,modified,completed)
values ('ebcf1850563c4de3b56813a52a95e930', 'Buy Movie Tickets','2018-08-17
09:50:10.126','2018-08-17 09:50:10.126',false);
insert into to_do (id,description,created,modified,completed)
values ('78269087206d472c894f3075031d8d6b', 'Clean my Room','2018-08-17
07:42:44.136','2018-08-17 07:42:44.137',false);
```

Теперь, если запустить наше приложение, вы должны увидеть в журналах:

```
Using generated security password: 2a569843-122a-4559-a245-60f5ab2b6c51
```

Это ваш пароль. Можете теперь перейти в браузер и открыть адрес http://localhost:8080/api/toDos. После нажатия Enter для перехода по этому адресу вы увидите что-то вроде изображенного на рис. 8.2.

Рис. 8.2. Приложение ToDo: страница http://localhost:8080/login

На рис. 8.2 показана страница входа в приложение — поведение по умолчанию при добавлении зависимости `spring-boot-starter-security`. По умолчанию

безопасность включена — так просто! Что ж, а какие имя пользователя и пароль? Как я уже упоминал ранее, имя пользователя — `user`, а паролем служит сгенерированный случайный пароль, выведенный в журнал (в данном примере `2a569843-122a-4559-a245-60f5ab2b6c51`). Итак, введите имя пользователя и пароль — и вы должны получить список запланированных дел (рис. 8.3).

Рис. 8.3. http://localhost:8080/api/toDos

Если же хотите воспользоваться командной строкой, можете выполнить следующую команду в окне терминала:

```
$ curl localhost:8080/api/toDos
{"timestamp":"2018-08-19T21:25:47.224+0000","status":401,"error":"Unauthori
 zed","message":"Unauthorized","path":"/api/toDos"}
```

Как вы видите из вывода в консоли, вам отказано в доступе к данной конечной точке. Необходима аутентификация, правда? Можете выполнить следующую команду:

```
$ curl localhost:8080/api/toDos -u user:2a569843-122a-4559-a245-60f5ab2b6c51
{
  "_embedded" : {
    "toDos" : [ {
      "description" : "Read a Book",
      "created" : "2018-08-17T07:42:44.136",
      "modified" : "2018-08-17T07:42:44.137",
      "completed" : true,
...
}
```

Как видите, теперь вы передаете имя пользователя и пароль и получаете ответ со списком запланированных дел.

Как вы, наверное, догадываетесь, при каждом перезапуске приложения автоконфигурация безопасности генерирует новый случайный пароль, что неоптимально и подходит, вероятно, лишь для разработки.

Переопределяем безопасность базового уровня

Случайные пароли не подходят для среды промышленной эксплуатации. Spring Security предоставляет множество вариантов переопределения настроек по умолчанию. Простейший способ — воспользоваться файлом `application.properties`, добавив в него следующие свойства `spring.security.*`:

```
spring.security.user.name=apress
spring.security.user.password=springboot2
spring.security.user.roles=ADMIN,USER
```

Если запустить приложение снова, именем пользователя для него будет `apress`, а паролем — `springboot2` (точно как в свойствах). Обратите внимание также, что в журнале более не выводится случайный пароль.

Можно также обеспечить аутентификацию программным образом. Создайте класс `ToDoSecurityConfig`, расширяющий класс `WebSecurityConfigurerAdapter`. Взгляните на листинг 8.4.

Листинг 8.4. com.apress.todo.config.ToDoSecurityConfig.java

```
package com.apress.todo.config;

import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.Configuration;
import  org.springframework.security.config.annotation.authentication.
  builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.configuration.
  WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
public class ToDoSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(
      AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication()
                    .passwordEncoder(passwordEncoder())
                    .withUser("apress")
                    .password(passwordEncoder().encode("springboot2"))
                    .roles("ADMIN","USER");
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

В листинге 8.4 приведены настройки, необходимые для построения безопасности программным образом, в данном случае с одним пользователем (конечно, можно добавить еще). Проанализируем этот код.

- Класс `WebSecurityConfigurerAdapter`. Наследование этого класса — один из способов переопределения настроек безопасности, поскольку позволяет переопределить методы, которые вам действительно нужны. В данном случае наш код переопределяет сигнатуру `configure(AuthenticationManagerBuilder)`.
- Класс `AuthenticationManagerBuilder`. Этот класс создает `AuthenticationManager`, с помощью которого можно легко осуществлять аутентификацию в оперативной памяти, аутентификацию LDAP, аутентификацию на основе JDBC, добавлять `UserDetailsService` и различные `AutheticationProvider`. В данном случае мы формируем процедуру аутентификации в оперативной памяти. Для более безопасного использования и шифрования/расшифровки пароля необходимо добавить `PasswordEncoder`.
- Класс `BCryptPasswordEncoder`. В этом коде используется объект `BCryptPasswordEncoder` (возвращает реализацию `PasswordEncoder`), который использует криптостойкую функцию хеширования. Можно также воспользоваться

`Pbkdf2PasswordEncoder` (использует стандарт формирования ключа PBKDF2 с настраиваемым числом итераций и случайным восьмибайтным начальным значением) или `SCryptPasswordEncoder` (использует функцию хеширования SCrypt). Или, еще лучше, задействуйте `DelegatingPasswordEncoder`, поддерживающий передачу обработки другому `PasswordEncoder` в зависимости от префикса.

Прежде чем запускать приложение, закомментируйте добавленные в файл `application.properties` свойства `spring.security.*`. Если запустить приложение, все должно работать так, как и ожидалось. Необходимо будет указать имя пользователя, `apress`, и пароль, `springboot2`.

Переопределение используемой по умолчанию страницы входа

Spring Security дает возможность переопределить используемую по умолчанию страницу входа несколькими способами. Один из них — настроить `HttpSecurity`. Класс `HttpSecurity` позволяет настраивать веб-безопасность для конкретных HTTP-запросов. По умолчанию он применяется ко всем запросам, но существует возможность ограничить охват с помощью `requestMatcher(RequestMatcher)` или аналогичных методов.

Взглянем на модифицированный класс `ToDoSecurityConfig` (листинг 8.5).

Листинг 8.5. com.apress.todo.config.ToDoSecurityConfig.java, версия 2

```
package com.apress.todo.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.
  builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.
  HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
  WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
public class ToDoSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
     protected void configure(AuthenticationManagerBuilder auth) throws
```

```
    Exception {
        auth.inMemoryAuthentication()
                .passwordEncoder(passwordEncoder())
                .withUser("apress")
                .password(passwordEncoder().encode("springboot2"))
                .roles("ADMIN","USER");
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .anyRequest().fullyAuthenticated()
                .and()
                .httpBasic();
    }
}
```

В листинге 8.5 приведена вторая версия класса `ToDoSecurityConfig`. Если теперь запустить приложение и перейти в браузер (http://localhost:8080/api/toDos), вы увидите всплывающее окно для простейшей аутентификации (рис. 8.4).

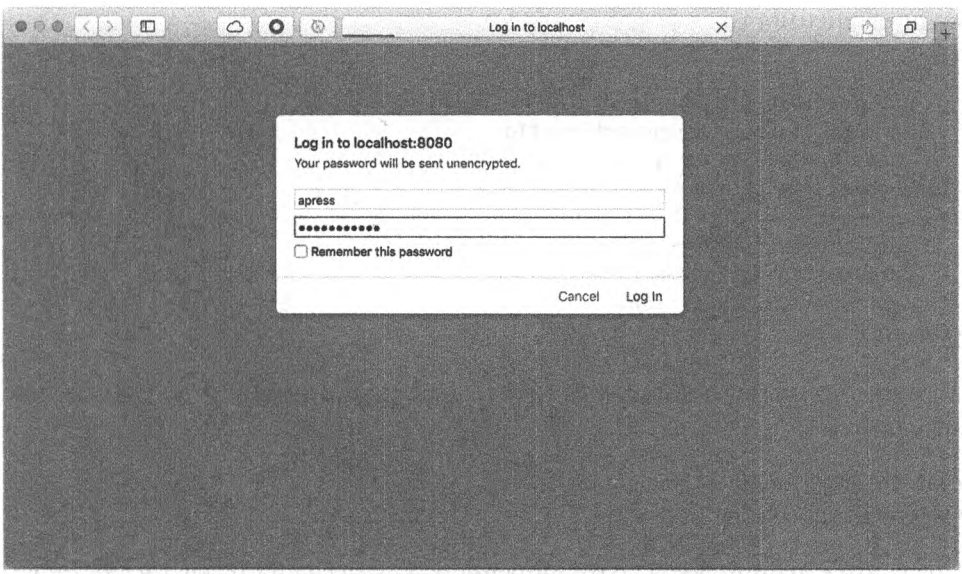


Рис. 8.4. Простейшая HTTP-аутентификация: http://localhost:8080/api/toDos

Можете применить уже известные вам имя пользователя и пароль и получить список запланированных дел. Точно так же и в командной строке. Необходимо указать учетные данные:

```
$ curl localhost:8080/api/toDos -u apress:springboot2
```

Пользовательская страница входа

Обычно в приложениях подобные страницы не увидишь; страница входа обычно очень аккуратна и хорошо спроектирована, правда? Spring Security позволяет создать и настроить по своему желанию собственную страницу входа.

Подготовим приложение ToDo со страницей входа. Во-первых, добавим CSS и широко известную библиотеку jQuery. В настоящее время в приложениях Spring Boot можно использовать зависимости WebJars. Этот новый способ позволяет избежать скачивания файлов вручную; вместо этого их можно указывать в качестве ресурсов. Автоконфигурация Spring Boot Web обеспечит необходимый доступ к ним.

Если вы используете Maven, откройте файл `pom.xml` и добавьте в него следующие зависимости:

```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>3.3.7</version>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>3.2.1</version>
</dependency>
```

Если вы применяете Gradle, откройте файл `build.gradle` и добавьте такие зависимости:

```
compile ('org.webjars:bootstrap:3.3.7')
compile ('org.webjars:jquery:3.2.1')
```

Создадим страницу входа с расширением `.mustache` (`login.mustache`). Ее необходимо создать в каталоге `src/main/resources/templates` (листинг 8.6).

Листинг 8.6. src/main/resources/templates/login.mustache

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial- scale=1">
    <title>ToDo's API Login Page</title>
    <link href="webjars/bootstrap/3.3.7/css/bootstrap.min.css" rel="stylesheet">
    <link href="css/signin.css" rel="stylesheet">
  </head>

  <body>

    <div class="container">
      <form class="form-signin" action="/login" method="POST">
        <h2 class="form-signin-heading">Please sign in</h2>
        <label for="username" class="sr-only">Username</label>
         <input type="text"     name="username" class="form-control"
             placeholder="Username" required autofocus>
        <label for="inputPassword" class="sr-only">Password</label>
         <input type="password" name="password" class="form-control"
             placeholder="Password" required>
         <button class="btn btn-lg btn-primary btn-block" id="login"
             type="submit">Sign in</button>
        <input type="hidden" name="_csrf" value="{{_csrf.token}}" />
      </form>
    </div>
  </body>
</html>
```

В листинге 8.6 приведена HTML-страница входа. В этой странице используется CSS из Bootstrap (https://getbootstrap.com) с помощью зависимостей WebJars (www.webjars.org). Данные файлы извлекаются из этих JAR в виде файловых ресурсов. В HTML-форме в качестве названий используются `username` и `password` (обязательное требование Spring Security). Во избежание возможных атак необходимо включить CSRF токен. Движок Mustache позволяет сделать это с помощью значения `{{_csrf.token}}`. Чтобы исключить возможность атак в запросах, Spring Security использует так называемый *паттерн токена синхронизатора* (synchronizer token pattern). Далее мы увидим, откуда берется это значение.

Создадим страницу index для просмотра домашней страницы и выхода из приложения. Создайте страницу `index.mustache` в каталоге `src/main/resources/templates` (листинг 8.7).

Листинг 8.7. src/main/resources/templates/index.mustache

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>ToDo's API</title>
     <link href="webjars/bootstrap/3.3.7/css/bootstrap.min.css"
         rel="stylesheet">
    <script src="webjars/jquery/3.2.1/jquery.min.js"></script>
</head>

<body>
<div class="container">
    <div class="header clearfix">
        <nav>
            <a href="#" id="logoutLink">Logout</a>
        </nav>
    </div>

    <div class="jumbotron">
        <h1>ToDo's Rest API</h1>
         <p class="lead">Welcome to the ToDo App. A Spring Boot
             application!</p>
    </div>
</div>

<form id="logout" action="/logout" method="POST">
    <input type="hidden" name="_csrf" value="{{_csrf.token}}" />
</form>
<script>
    $(function(){
        $('#logoutLink').click(function(){
            $('#logout').submit();
        });
    });
</script>
</body>
</html>
```

В листинге 8.7 показана начальная страница index. Мы по-прежнему используем ресурсы Bootstrap и jQuery, а главное — `{{_csrf.token}}` для выхода из приложения.

Займемся конфигурацией. Во-первых, необходимо модифицировать класс `ToDoSecurityConfig` (листинг 8.8).

Листинг 8.8. com.apress.todo.config.ToDoSecurityConfig.java — v3

```
package com.apress.todo.config;

import org.springframework.boot.autoconfigure.security.servlet.PathRequest;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders.
  AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
  WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.web.util.matcher.AntPathRequestMatcher;

@Configuration
public class ToDoSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
     protected void configure(AuthenticationManagerBuilder auth) throws
        Exception {
        auth.inMemoryAuthentication()
                .passwordEncoder(passwordEncoder())
                .withUser("apress")
                .password(passwordEncoder().encode("springboot2"))
                .roles("ADMIN","USER");
    }

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .requestMatchers(
                        PathRequest
                              .toStaticResources()
                              .atCommonLocations()).permitAll()
                .anyRequest().fullyAuthenticated()
                .and()
                .formLogin().loginPage("/login").permitAll()
                .and()
                .logout()
                    .logoutRequestMatcher(
                        new AntPathRequestMatcher("/logout"))
                    .logoutSuccessUrl("/login");
    }
}
```

В листинге 8.8 приведена версия 3 класса `ToDoSecurityConfig`. В этой новой версии показывается процесс настройки `HttpSecurity`. Во-первых, добавляется метод `requestMatchers`, указывающий на общие места, где хранятся, например, статические ресурсы (`static/*`). Именно там располагаются CSS, JS и другие простые HTML-файлы, не требующие обеспечения безопасности. Далее используется `anyRequest` и `fullyAuthenticated`, а это значит, что аутентификации требуют и `/api/*`. После задействуется метод `formLogin`, чтобы указать с помощью вызова `loginPage("/login")`, где искать конечную точку для страницы входа. Объявляется конечная точка выхода из приложения (`"/logout"`); в случае успешного выхода пользователь перенаправляется на конечную точку/страницу `"/login"`.

Теперь нужно указать Spring MVC, где искать страницу входа. Создайте класс `ToDoWebConfig` (листинг 8.9).

Листинг 8.9. com.apress.todo.config.ToDoWebConfig.java

```
package com.apress.todo.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.
  ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class ToDoWebConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/login").setViewName("login");
    }
}
```

В листинге 8.9 показан другой способ настройки веб-контроллера в Spring MVC. При этом по-прежнему можно использовать класс, снабженный аннотацией `@Controller`, задавая соответствие для страницы входа; но это вариант JavaConfig.

Тут же класс реализует интерфейс `WebMvcConfigure`. В нем реализован метод `addViewControllers`, в котором регистрируется конечная точка `/login` с помощью указания контроллеру на местоположение соответствующего представления. В результате он найдет страницу `templates/login.mustache`.

Наконец, необходимо отредактировать файл `application.properties`, добавив в него следующее свойство:

```
spring.mustache.expose-request-attributes=true
```

Помните токен `{{_csrf.token}}`? Именно таким образом он получает значение — с помощью добавления свойства `spring.mustache.expose-request-attributes`.

Теперь можно запустить приложение. Если вы перейдете по адресу http://localhost:8080, то увидите что-то вроде изображенного на рис. 8.5.

Рис. 8.5. http://localhost:8080/login

Вам была выдана пользовательская страница входа. Замечательно! Можете ввести здесь учетные данные и увидите начальную страницу (рис. 8.6).

После получения домашней страницы можете посетить адрес http://localhost:8080/api/toDos. Поскольку вы полностью аутентифицированы, то можете вернуться к списку запланированных дел. Или вернуться на домашнюю страницу и щелкнуть на ссылке Logout (Выход из приложения), которая перенаправит вас опять на конечную точку `/login`.

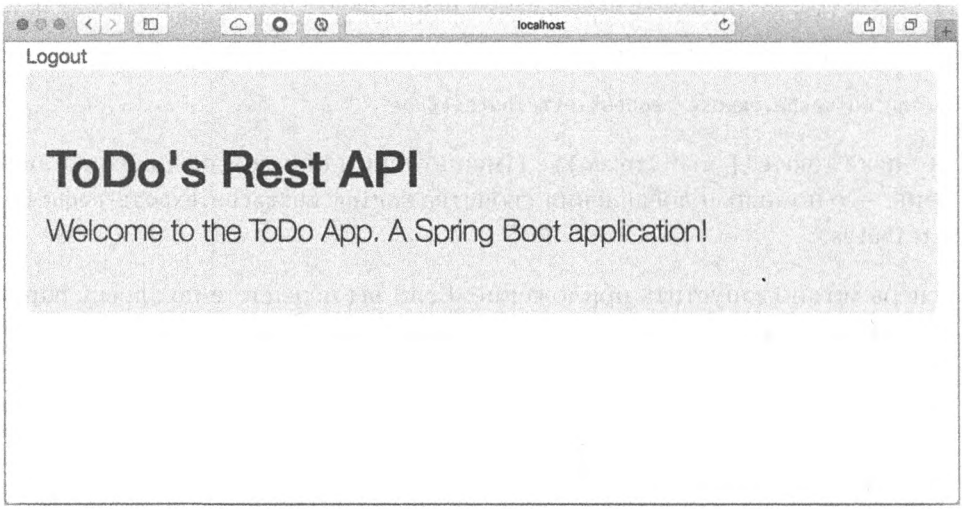


Рис. 8.6. http://localhost:8080 после входа в приложение

Теперь разберемся, что произойдет, если попытаться выполнить следующую команду в окне терминала:

```
$ curl localhost:8080/api/toDos -u apress:springboot2
```

Она не вернет ничего, только пустую строку. Если воспользоваться флагом `-i`, будет выведена информация о перенаправлении на http://localhost:8080/login. Но возможности взаимодействовать с формой из командной строки нет, правда? Так как же решить эту проблему? На практике существуют клиенты, вообще не использующие веб-интерфейсы. Большинство из клиентов — другие приложения, которые должны использовать API REST программным образом, но в нашем текущем решении не существует способа аутентификации для взаимодействия с формой.

Откройте класс `ToDoSecurityConfig` и измените метод `configure(HttpSecurity)` так, как показано в следующем фрагменте кода:

```
@Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
                .requestMatchers(PathRequest.toStaticResources().
                     atCommonLocations()).permitAll()
                .anyRequest().fullyAuthenticated()
                .and()
                .formLogin().loginPage("/login").permitAll()
```

```
            .and()
            .logout()
                .logoutRequestMatcher(
                      new AntPathRequestMatcher("/logout"))
                .logoutSuccessUrl("/login")
            .and()
                        .httpBasic();
    }
```

В последних двух строках метода мы добавляем вызов `httpBasic`, позволяющий клиентам (например, cURL) использовать базовые механизмы аутентификации. Можете перезапустить приложение ToDo и увидите, что вышеупомянутая команда работает теперь в командной строке.

Безопасность при использовании JDBC

Представьте себе, что у вашей компании есть база данных сотрудников, которую вы хотели использовать для аутентификации и авторизации в приложении ToDo. Было бы неплохо интегрировать в свое приложение нечто подобное, правда?

Spring Security дает возможность использования `AuthenticationManager` с механизмами аутентификации в оперативной памяти, LDAP и JDBC. В этом разделе мы модифицируем приложение ToDo для работы с JDBC.

Создание приложения-справочника с использованием средств безопасности JDBC

В этом разделе мы создадим новое приложение — приложение-справочник персонала фирмы. Приложение Directory интегрировано в приложение ToDo для решения задач аутентификации и авторизации. Так, если клиент хочет добавить новое запланированное дело, он должен быть аутентифицирован с ролью `USER`.

Приступим. Если вы начинаете с нуля, перейдите в браузер и откройте сайт Spring Initializr (https://start.spring.io). Внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.directory`.
- Artifact (Артефакт): `directory`.

- Name (Название): `directory`.
- Package Name (Название пакета): `com.apress.directory`.
- Dependencies (Зависимости): `Web`, `Security`, `Lombok`, `JPA`, `REST Repositories`, `H2`, `MySQL`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в вашу любимую IDE (рис. 8.7).

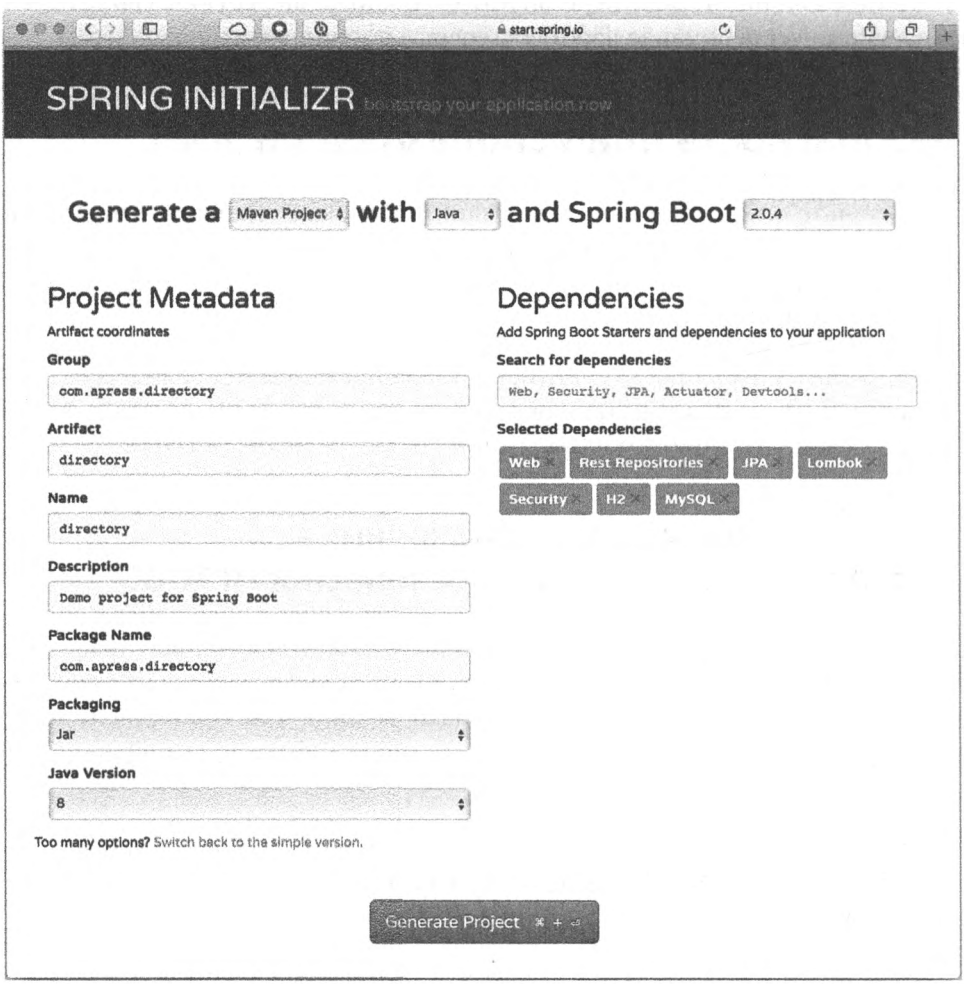


Рис. 8.7. Spring Initializr

Как вы видите, список зависимостей очень похож на список для остальных наших приложений. Мы воспользуемся возможностями Spring Data, Spring Security и REST. Начнем с добавления нового класса для хранения информации о конкретном человеке. Создайте класс `Person` (листинг 8.10).

Листинг 8.10. com.apress.directory.domain.Person.java

```
package com.apress.directory.domain;

import lombok.Data;
import org.hibernate.annotations.GenericGenerator;
import javax.persistence.*;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
@Data
@Entity
public class Person {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    @Column(unique = true)
    private String email;
    private String name;
    private String password;
    private String role = "USER";
    private boolean enabled = true;
    private LocalDate birthday;

    @Column(insertable = true, updatable = false)
    private LocalDateTime created;
    private LocalDateTime modified;

    public Person() {
    }

     public Person(String email, String name, String password, String birthday) {
        this.email = email;
        this.name = name;
        this.password = password;
        this.birthday = LocalDate.parse(birthday, DateTimeFormatter.
            ofPattern("yyyy-MM-dd"));
    }

     public Person(String email, String name, String password, LocalDate
        birthday) {
        this.email = email;
```

```
        this.name = name;
        this.password = password;
        this.birthday = birthday;
    }

     public Person(String email, String name, String password, String
        birthday, String role, boolean enabled) {
        this(email, name, password, birthday);
        this.role = role;
        this.enabled = enabled;
    }

    @PrePersist
    void onCreate() {
        this.setCreated(LocalDateTime.now());
        this.setModified(LocalDateTime.now());
    }

    @PreUpdate
    void onUpdate() {
        this.setModified(LocalDateTime.now());
    }
}
```

В листинге 8.10 приведен класс `Person`, очень простой. Он включает достаточную информацию о человеке. Далее создадим репозиторий — интерфейс `PersonRepository` (листинг 8.11).

Листинг 8.11. com.apress.directory.repository.PersonRepository.java

```
package com.apress.directory.repository;

import com.apress.directory.domain.Person;
import org.springframework.data.repository.CrudRepository;
import org.springframework.data.repository.query.Param;

public interface PersonRepository extends CrudRepository<Person,String> {
    public Person findByEmailIgnoreCase(@Param("email") String email);
}
```

В листинге 8.11 приведен интерфейс `PersonRepository`; но чем он отличается от остальных наших репозиториев? Он объявляет метод-запрос `findByEmailIgnoreCase` с адресом электронной почты в качестве параметра (снабженного аннотацией `@Param`). Такой синтаксис указывает Spring Data REST на необходимость реализации этих методов и создания соответствующего SQL-оператора (на основе названия и полей из класса предметной области; в данном случае поля `email`).

ПРИМЕЧАНИЕ

Если хотите узнать больше о том, как описывать методы-запросы, взгляните на справочную документацию Spring Data JPA по адресу https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods.

Создайте класс `DirectorySecurityConfig`, наследующий класс `WebSecurityConfigurerAdapter`. Помните, что с помощью наследования этого класса можно настраивать под свои нужды Spring Security для данного приложения (листинг 8.12).

Листинг 8.12. com.apress.directory.config.DirectorySecurityConfig.java

```
package com.apress.directory.config;

import com.apress.directory.repository.PersonRepository;
import com.apress.directory.security.DirectoryUserDetailsService;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.config.annotation.authentication.builders.
  AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
  WebSecurityConfigurerAdapter;

@Configuration
public class DirectorySecurityConfig extends WebSecurityConfigurerAdapter {

    private PersonRepository personRepository;

    public DirectorySecurityConfig(PersonRepository personRepository){
        this.personRepository = personRepository;
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
                .authorizeRequests()
                .antMatchers("/**").hasRole("ADMIN")
                .and()
                .httpBasic();
    }

    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.userDetailsService(
            new DirectoryUserDetailsService(this.personRepository));
    }
}
```

В листинге 8.12 приведен класс `DirectorySecurityConfig`. Он задает настройки `HttpSecurity`, разрешающие при базовой аутентификации доступ к любой из конечных точек (`/**`) только пользователям с ролью `ADMIN`.

Что еще отличается от других наших настроек безопасности? Да, вы правы! `AuthenticationManager` задает настройки реализации `UserDetailsService`. Это ключ к использованию любых сторонних приложений для обеспечения безопасности и интеграции их со Spring Security.

Как вы видите, метод `UserDetailsService` использует класс `DirectoryUserDetailsService`. Создадим этот класс (листинг 8.13).

Листинг 8.13. com.apress.directory.security.DirectoryUserDetailsService.java

```
package com.apress.directory.security;

import com.apress.directory.domain.Person;
import com.apress.directory.repository.PersonRepository;
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.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;

public class DirectoryUserDetailsService implements UserDetailsService {

    private PersonRepository repo;

    public DirectoryUserDetailsService(PersonRepository repo) {
        this.repo = repo;
    }

    @Override
    public UserDetails loadUserByUsername(String username) throws
        UsernameNotFoundException {
        try {
            final Person person = this.repo.findByEmailIgnoreCase(username);

            if (person != null) {
                PasswordEncoder encoder = PasswordEncoderFactories.
                    createDelegatingPasswordEncoder();
                String password = encoder.encode(person.getPassword());

                return User.withUsername(person.getEmail()).
                    accountLocked(!person.isEnabled()).
                    password(password).roles(person.getRole()).build();
            }
```

```
        }catch(Exception ex){
            ex.printStackTrace();
        }
        throw new UsernameNotFoundException(username);
    }
}
```

В листинге 8.13 приведен класс `DirectoryUserDetailsService`. Он реализует интерфейс `UserDetailsService` и должен реализовать метод `loadUserByUsername` и возвращать экземпляр `UserDetails`. В этой реализации код демонстрирует, как используется `PersonRepository`. В данном случае задействуется метод `findByEmailIgnoreCase`; так, если при запросе пользователем доступа к `/**` (любой конечной точке) найден человек с указанным адресом электронной почты, он проверяет соответствие электронной почты указанному паролю, роли и, в зависимости от того, заблокирована ли учетная запись, создает экземпляр `UserDetails`.

Потрясающе! Данное приложение использует в качестве механизма аутентификации JDBC. Опять же, можно подключить для этой цели любую другую систему/приложение обеспечения безопасности, реализующее интерфейс `UserDetailsService` и возвращающее экземпляр `UserDetails`, вот и все.

Взглянем на файл `application.properties` и на содержащиеся в нем свойства:

```
# Server
server.port=${port:8181}

# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true

# H2
spring.h2.console.enabled=true
```

Единственное отличие заключается в свойстве `server.port`, которое гласит: если будет указана переменная для порта (в командной строке или среде), будет использоваться она; если же нет, будет использоваться порт 8181. Это выражение — составная часть языка выражений Spring (SpEL).

Прежде чем запускать приложение Directory, добавим в него данные. Создайте файл `data.sql` в каталоге `src/main/resources`.

```
insert into person (id,name,email,password,role,enabled,birthday,created,
 modified)
values ('dc952d19ccfc4164b5eb0338d14a6619','Mark','mark@example.com',
```

```
 'secret','USER',true,'1960-03-29','2018-08-17 07:42:44.136',
 '2018-08-17 07:42:44.137');
insert into person (id,name,email,password,role,enabled,birthday,created,
 modified)
values ('02288a3b194e49ceb1803f27be5df457','Matt','matt@example.com',
 'secret','USER',true,'1980-07-03','2018-08-17 07:42:44.136',
 '2018-08-17 07:42:44.137');
insert into person (id,name,email,password,role,enabled,birthday,created,
 modified)
values ('4fe22e358d0e4e38b680eab91787f041','Mike','mike@example.com',
 'secret','ADMIN',true,'19820-08-05','2018-08-17 07:42:44.136',
 '2018-08-17 07:42:44.137');
insert into person (id,name,email,password,role,enabled,birthday,created,
 modified)
values ('84e6c4776dcc42369510c2692f129644','Dan','dan@example.com',
 'secret','ADMIN',false,'1976-10-11','2018-08-17 07:42:44.136',
 '2018-08-17 07:42:44.137');
insert into person (id,name,email,password,role,enabled,birthday,created,
 modified)
values ('03a0c396acee4f6cb52e3964c0274495','Administrator',
 'admin@example.com','admin','ADMIN',true,'1978-12-22',
 '2018-08-17 07:42:44.136','2018-08-17 07:42:44.137');
```

Теперь можно использовать это приложение в качестве механизма аутентификации и авторизации. Запустите приложение Directory. Оно запустится на порте 8181. Можете протестировать его с помощью браузера и/или команды cURL.

```
$ curl localhost:8181/persons/search/findByEmailIgnoreCase?email=mark@
example.com  -u admin@example.com:admin
{
  "email" : "mark@example.com",
  "name" : "Mark",
  "password" : "secret",
  "role" : "USER",
  "enabled" : true,
  "birthday" : "1960-03-29",
  "created" : "2018-08-17T07:42:44.136",
  "modified" : "2018-08-17T07:42:44.137",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8181/persons/
          dc952d19ccfc4164b5eb0338d14a6619"
    },
    "person" : {
      "href" : "http://localhost:8181/persons/
          dc952d19ccfc4164b5eb0338d14a6619"
    }
  }
}
```

С помощью этой команды мы получаем информацию о пользователе Mark, указывая имя/пароль пользователя с ролью `ADMIN`; в данном случае с помощью параметра `-u admin@example.com:admin`.

Отлично! Мы научились использовать JDBC для поиска пользователей в справочнике с помощью Spring Data REST и Spring Security! Можете оставить это приложение запущенным.

Использование приложения Directory в приложении ToDo

Пора интегрировать приложение Directory с приложением ToDo. И сделать это очень просто.

Откройте приложение ToDo и создайте класс `Person`. Да, нам понадобится класс `Person`, в котором будет содержаться ровно столько информации, сколько нужно для целей аутентификации и авторизации. Хранить в нем дату рождения или какую-либо другую информацию не нужно (листинг 8.14).

Листинг 8.14. com.apress.todo.directory.Person.java

```
package com.apress.todo.directory;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import lombok.Data;

@Data
@JsonIgnoreProperties(ignoreUnknown = true)
public class Person {

    private String email;
    private String password;
    private String role;
    private boolean enabled;
}
```

В листинге 8.14 приведен класс `Person`. Он содержит только поля, необходимые для процесса аутентификации и авторизации. Важно отметить, что при обращении к приложению Directory возвращается объект в формате JSON, содержащий более полную информацию. Они должны соответствовать для десериализации (из JSON в объект с помощью библиотеки Jackson), но, поскольку необходимости в дополнительной информации нет, в классе используется аннотация `@JasonIgnoreProperties(ignoreUnknown=true)` для упрощения

сопоставления полей. Мне кажется, что это отличный способ расцепления классов.

ПРИМЕЧАНИЕ

Для некоторых утилит десериализации в Java необходим тот же класс в том же пакете, причем реализующий java.io.Serializable, что сильно затрудняет его расширение и вообще работу с ним разработчикам и клиентам.

Создадим класс `ToDoProperties` для хранения информации о приложении Directory, например `Uri` (адрес и базовый URI), `Username` и `Password` человека с ролью `ADMIN` и доступом к REST API (листинг 8.15).

Листинг 8.15. com.apress.todo.config.ToDoProperties.java

```
package com.apress.todo.config;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

@Data
@ConfigurationProperties(prefix = "todo.authentication")
public class ToDoProperties {

    private String findByEmailUri;
    private String username;
    private String password;
}
```

В листинге 8.15 приведен класс `ToDoProperties`; обратите внимание на префикс `todo.authentication.*`. Скопируйте класс `ToDoSecurityConfig`. Можете закомментировать весь класс и скопировать код из листинга 8.16.

Листинг 8.16. com.apress.todo.config.ToDoSecurityConfig.java

```
package com.apress.todo.config;

import com.apress.todo.directory.Person;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.autoconfigure.security.servlet.PathRequest;
import org.springframework.boot.context.properties.
  EnableConfigurationProperties;
import org.springframework.boot.web.client.RestTemplateBuilder;
```

```
import org.springframework.context.annotation.Configuration;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.hateoas.MediaTypes;
import org.springframework.hateoas.Resource;
import org.springframework.http.HttpStatus;
import org.springframework.http.RequestEntity;
import org.springframework.http.ResponseEntity;
import org.springframework.security.config.annotation.authentication.builders.
  AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.
  HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
  WebSecurityConfigurerAdapter;
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.security.crypto.factory.PasswordEncoderFactories;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.util.matcher.AntPathRequestMatcher;
import org.springframework.web.client.RestTemplate;
import org.springframework.web.util.UriComponentsBuilder;
import java.net.URI;

@EnableConfigurationProperties(ToDoProperties.class)
@Configuration
public class ToDoSecurityConfig extends WebSecurityConfigurerAdapter {

     private final Logger log = LoggerFactory.getLogger(ToDoSecurityConfig.class);

    // Это можно использовать для подключения к приложению Directory
    private RestTemplate restTemplate;
    private ToDoProperties toDoProperties;
    private UriComponentsBuilder builder;

    public ToDoSecurityConfig(RestTemplateBuilder restTemplateBuilder,
        ToDoProperties toDoProperties){
        this.toDoProperties = toDoProperties;
        this.restTemplate = restTemplateBuilder.basicAuthorization(
            toDoProperties.getUsername(),toDoProperties.getPassword()).
            build();
    }

    @Override
     protected void configure(AuthenticationManagerBuilder auth) throws
        Exception {
        auth.userDetailsService(new UserDetailsService(){

            @Override
            public UserDetails loadUserByUsername(String username) throws
```

```
            UsernameNotFoundException {
              try {
                builder = UriComponentsBuilder
                .fromUriString(toDoProperties.getFindByEmailUri())
                    .queryParam("email", username);
                log.info("Querying: " + builder.toUriString());
                ResponseEntity<Resource<Person>> responseEntity =
                restTemplate.exchange(
                    RequestEntity.get(URI.create(builder.toUriString()))
                                        .accept(MediaTypes.HAL_JSON)
                                        .build()
                       , new ParameterizedTypeReference<Resource
                       <Person>>() {
                       });

              if (responseEntity.getStatusCode() == HttpStatus.OK) {

                  Resource<Person> resource = responseEntity.getBody();
                  Person person = resource.getContent();

                  PasswordEncoder encoder =
                  PasswordEncoderFactories.createDelegatingPasswordEncoder();
                  String password = encoder.encode(person.getPassword());
                  return User
                  .withUsername(person.getEmail())
                  .password(password)
                  .accountLocked(!person.isEnabled())
                  .roles(person.getRole()).build();
                  }
              }catch(Exception ex) {
                  ex.printStackTrace();
              }
              throw new UsernameNotFoundException(username);
          }
      });
  }

  @Override
  protected void configure(HttpSecurity http) throws Exception {
      http.authorizeRequests()
              .requestMatchers(PathRequest.toStaticResources().
                   atCommonLocations()).permitAll()
              .antMatchers("/","/api/**").hasRole("USER")
              .and()
              .formLogin().loginPage("/login").permitAll()
              .and()
              .logout()
```

```
                .logoutRequestMatcher(new AntPathRequestMatcher("/logout"))
                .logoutSuccessUrl("/login")
                .and()
                .httpBasic();
    }
}
```

В листинге 8.16 приведен новый класс `ToDoSecurityConfig`. Проанализируем его.

- Класс `WebSecurityConfigurerAdapter`. Этот класс переопределяет все необходимое для настройки безопасности приложения под наши нужды; но вы это уже знаете, правда?
- Класс `RestTemplate`. Этот вспомогательный класс выполняет REST-вызов к конечной точке приложения Directory, а именно к URI `/persons/search/findByEmailIgnoreCase`.
- Класс `UriComponentsBuilder`. Напомню, что для конечной точки `/persons/search/findByEmailIgnoreCase` требуется параметр (`email`); его предоставляет метод `loadUserByUsername` (`username`).
- Класс `AuthenticationManagerBuilder`. Аутентификацию обеспечивает `UserDetailsService`. В этом коде можно увидеть анонимную реализацию `UserDetailsService` и реализацию метода `loadUserByUsername`. Именно там используется `RestTemplate` для обращения к приложению Directory и конечной точке.
- `ResponseEntity`. Поскольку приложение Directory дает ответ в формате `HAL+JSON`, необходимо воспользоваться объектом `ResponseEntity`, который берет на себя работу со всеми ресурсами протокола. При `HttpStatus.OK` можно легко получить контент в виде экземпляра `Person` и создать на его основе `UserDetails`.
- `antMatchers`. Наш класс настраивает `HttpSecurity` так же, как и раньше, но на этот раз включает вызов метода `antMatchers`, отображающий конечные точки при обращении к ним допустимого пользователя с ролью `USER`.

Мы переиспользуем ту же методику, что и в приложении Directory. `AuthenticationManager` настраивается для получения объекта `UserDetails` с помощью вызова сервиса справочника (нашего приложения Directory) посредством `RestTemplate`. Приложение Directory возвращает ответ в формате HAL+JSON, поэтому для получения информации о человеке в виде ресурса необходимо воспользоваться `ResponseEntity`.

Добавьте следующие свойства `todo.authentication.*` в конец файла `application.properties`.

```
# ToDo – интеграция приложения Directory
todo.authentication.find-by-email-uri=http://localhost:8181/persons/search/
  findByEmailIgnoreCase
todo.authentication.username=admin@example.com
todo.authentication.password=admin
```

Необходимо, чтобы у пользователя была роль **`ADMIN`**, а также нужно указать полный URI для поиска конечной точки электронной почты.

Теперь мы готовы к применению приложения ToDo. Можете воспользоваться браузером или командной строкой. Проверьте, что приложение Directory запущено. Запустите приложение ToDo, работающее на порте 8080.

Можете выполнить следующую команду в окне терминала:

```
$ curl localhost:8080/api/toDos -u mark@example.com:secret
{
  "_embedded" : {
    "toDos" : [ {
      "description" : "Read a Book",
      "created" : "2018-08-17T07:42:44.136",
      "modified" : "2018-08-17T07:42:44.137",
      "completed" : true,
...
...
"profile" : {
      "href" : "http://localhost:8080/api/profile/toDos"
    }
  }
}
```

Теперь вы аутентифицированы и авторизованы как Mark с ролью `USER`. Мои поздравления! Вы интегрировали свой собственный сервис JDBC с приложением ToDo.

Безопасность WebFlux

Добавление средств безопасности в приложение WebFlux производится совершенно аналогично. Необходимо добавить зависимость `spring-boot-starter-security`, а Spring Boot со своей автоконфигурацией позаботится обо всем остальном. Для настройки под свои нужды, подобно тому как мы делали раньше, понадобится только воспользоваться `ReactiveUserDetailsService`

(вместо `UserDetailsService`) или `ReactiveAuthenticationManager` (вместо `AuthenticationManager`). Помните, что теперь вы работаете с типами реактивных потоков данных Mono и Flux.

Создание приложения ToDo с OAuth2

При использовании Spring Boot и Spring Security реализация OAuth2 упрощается до предела. В этом разделе данной главы мы выполним вход непосредственно в приложение ToDo с помощью OAuth2. Я предполагаю, что вы знакомы с OAuth2 и со всеми преимуществами использования его в качестве механизма аутентификации с помощью сторонних поставщиков — таких как Google, Facebook и GitHub — непосредственно в вашем приложении.

Итак, приступим. Если вы начинаете с нуля, то перейдите в браузер и откройте сайт Spring Initializr (https://start.spring.io). Внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-oauth2`.
- Name (Название): `todo-oauth2`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Web`, `Security`, `Lombok`, `JPA`, `REST Repositories`, `H2`, `MySQL`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в вашу любимую IDE (рис. 8.8).

Если вы используете Maven, откройте файл `pom.xml` и добавьте в него следующие зависимости:

```
<dependency>
     <groupId>org.springframework.security</groupId>
     <artifactId>spring-security-oauth2-client</artifactId>
</dependency>
<dependency>
     <groupId>org.springframework.security</groupId>
     <artifactId>spring-security-oauth2-jose</artifactId>
</dependency>
```

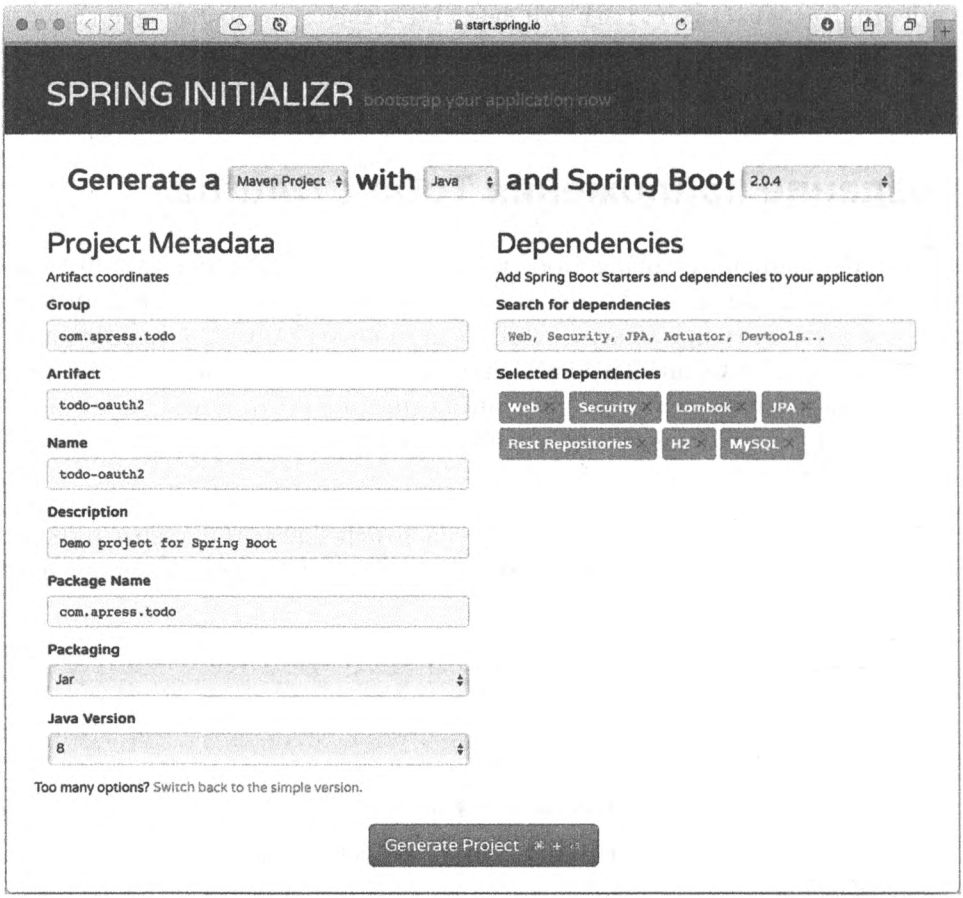


Рис. 8.8. Spring Initializr

Если вы используете Gradle, откройте файл `build.gradle` и добавьте следующие зависимости:

```
compile('org.springframework.security:spring-security-oauth2-client')
compile('org.springframework.security:spring-security-oauth2-jose')
```

Как вы догадываетесь, Spring Boot, обнаружив зависимость `spring-security-oauth2-client`, автоматически настраивает все необходимые компоненты для использования средств безопасности OAuth2 для данного приложения. Важно отметить потребность в зависимости `spring-security-oauth2-jose`, содержащей средства поддержки Spring Security для фреймворка JOSE

(JavaScript Object Signing and Encryption, «подписи и шифрование объектов JavaScript»). Фреймворк JOSE служит для безопасной передачи утверждений между участвующими сторонами. Он основан на следующих наборах спецификаций: JSON Web Token (JWT), JSON Web Signature (JWS), JSON Web Encryption (JWE) и JSON Web Key (JWK).

Далее вы можете переиспользовать класс `ToDo` и интерфейс `ToDoRepository` (листинги 8.17 и 8.18).

Листинг 8.17. com.apress.todo.domain.ToDo.java

```
package com.apress.todo.domain;

import lombok.Data;
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;

@Entity
@Data
public class ToDo {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    @NotNull
    @NotBlank
    private String description;

    @Column(insertable = true, updatable = false)
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){}
    public ToDo(String description){
        this.description = description;
    }

    @PrePersist
    void onCreate() {
        this.setCreated(LocalDateTime.now());
        this.setModified(LocalDateTime.now());
    }
```

```
    @PreUpdate
    void onUpdate() {
        this.setModified(LocalDateTime.now());
    }
}
```

Как видите, ничего не поменялось. Класс остался таким же, как был.

Листинг 8.18. com.apress.todo.repository.ToDoRepository.java

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends CrudRepository<ToDo,String> { }
```

И интерфейс тоже не поменялся. Взглянем на файл `application.properties`.

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true

# H2-Console: http://localhost:8080/h2-console
# jdbc:h2:mem:testdb
spring.h2.console.enabled=true
```

Ничего не поменялось. Что ж, очень скоро мы добавим в него дополнительные свойства.

А теперь — самое главное. Сейчас вам предстоит воспользоваться GitHub с целью OAuth2-аутентификации для приложения ToDo.

Создание приложения ToDo в GitHub

Я предполагаю, что у вас уже есть учетная запись в GitHub, если нет — можете легко создать новую на сайте https://github.com. Войдите в вашу учетную запись и откройте страницу https://github.com/settings/applications/new. Именно здесь вы и создадите приложение. Можете использовать следующие значения.

- Application name (Название приложения): `todo-app`.
- Homepage URL (URL домашней страницы): http://localhost:8080.

- Application description (Описание приложения): ToDo App.
- Authorization callback URL (URL обратного вызова авторизации): http://localhost:8080/login/oauth2/code/github.

Важно указать этот URL обратного вызова авторизации, поскольку именно так объект `OAuth2LoginAuthenticationFilter` собирается работать с этим паттерном конечной точки: `/login/oauth2/code/*`; конечно, все это можно настроить с помощью свойства `redirect-uri-template` (рис. 8.9).

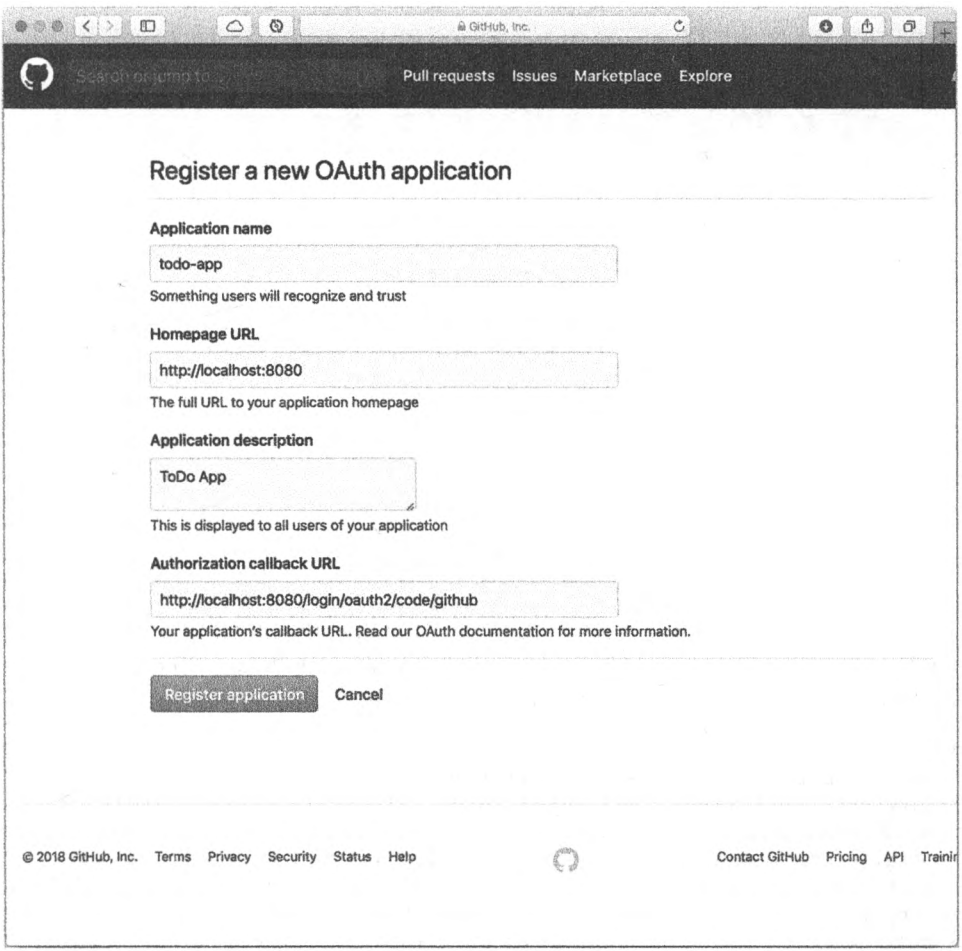


Рис. 8.9. Новое приложение GitHub: https://github.com/settings/applications/new

Можете нажать кнопку Register application (Зарегистрировать приложение). После этого GitHub создаст ключи, необходимые для нашего приложения (рис. 8.10).

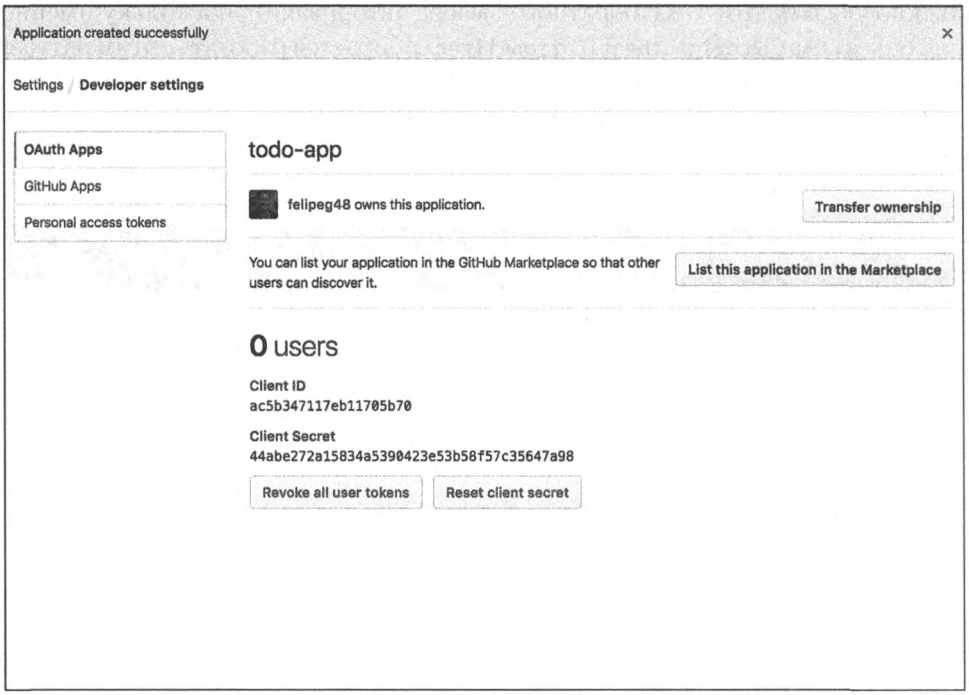


Рис. 8.10. Идентификатор и секретные ключи клиента

После этого скопируйте идентификатор и секретные ключи клиента и добавьте их в конец файла `application.properties` с помощью ключей `spring.security.oauth2.client.registration.*`.

```
# OAuth2
spring.security.oauth2.client.registration.todo.client-id=ac5b347117eb11705b70
spring.security.oauth2.client.registration.todo.client-secret=44abe272a1583
  4a5390423e53b58f57c35647a98
spring.security.oauth2.client.registration.todo.client-name=ToDo App with
  GitHub Authentication
spring.security.oauth2.client.registration.todo.provider=github
spring.security.oauth2.client.registration.todo.scope=user
spring.security.oauth2.client.registration.todo.redirect-uri-template=http://
  localhost:8080/login/oauth2/code/github
```

Свойство `spring.security.oauth2.client.registration` принимает в качестве значения ассоциативный массив, содержащий необходимые ключи, например `client-id` и `client-secret`.

Вот и все! Больше ничего не требуется. Можете запустить приложение. Откройте браузер и перейдите по адресу http://localhost:8080. Вы получите ссылку, которая перенаправит вас в GitHub (рис. 8.11).

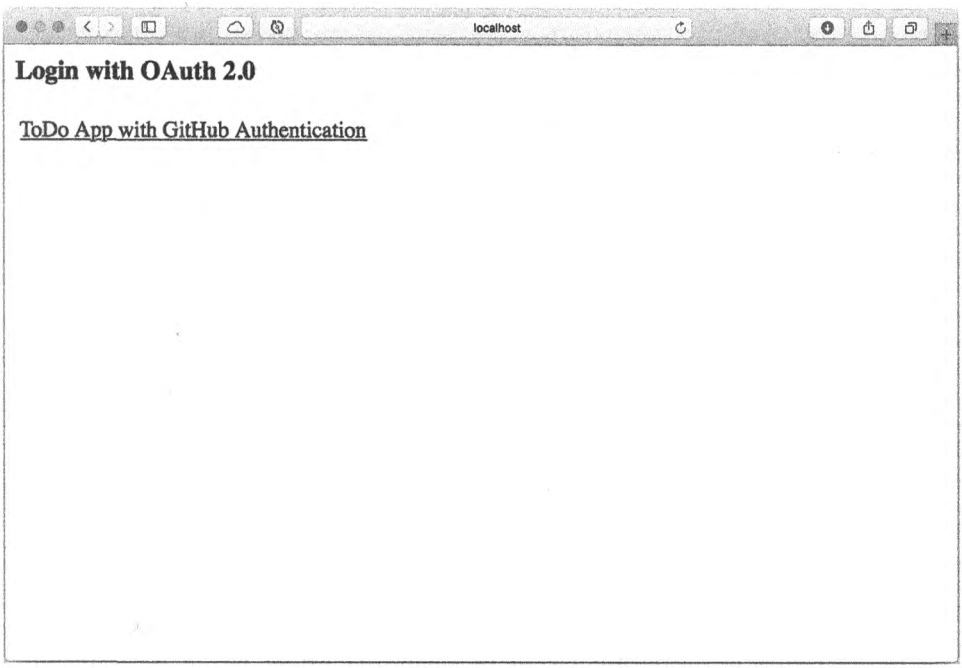


Рис. 8.11. http://localhost:8080

Можете щелкнуть на этой ссылке, которая проведет вас по процессу входа в приложение, но с использованием механизма аутентификации GitHub (рис. 8.12).

Теперь можете войти в приложение со своими учетными данными. Вы будете перенаправлены на другую страницу, где необходимо будет дать разрешение приложению *todo-app* использовать ваши контактные данные (рис. 8.13).

Далее нажмите кнопку Authorize (Авторизовать), чтобы вернуться в свое приложение с API REST ToDo (рис. 8.14).

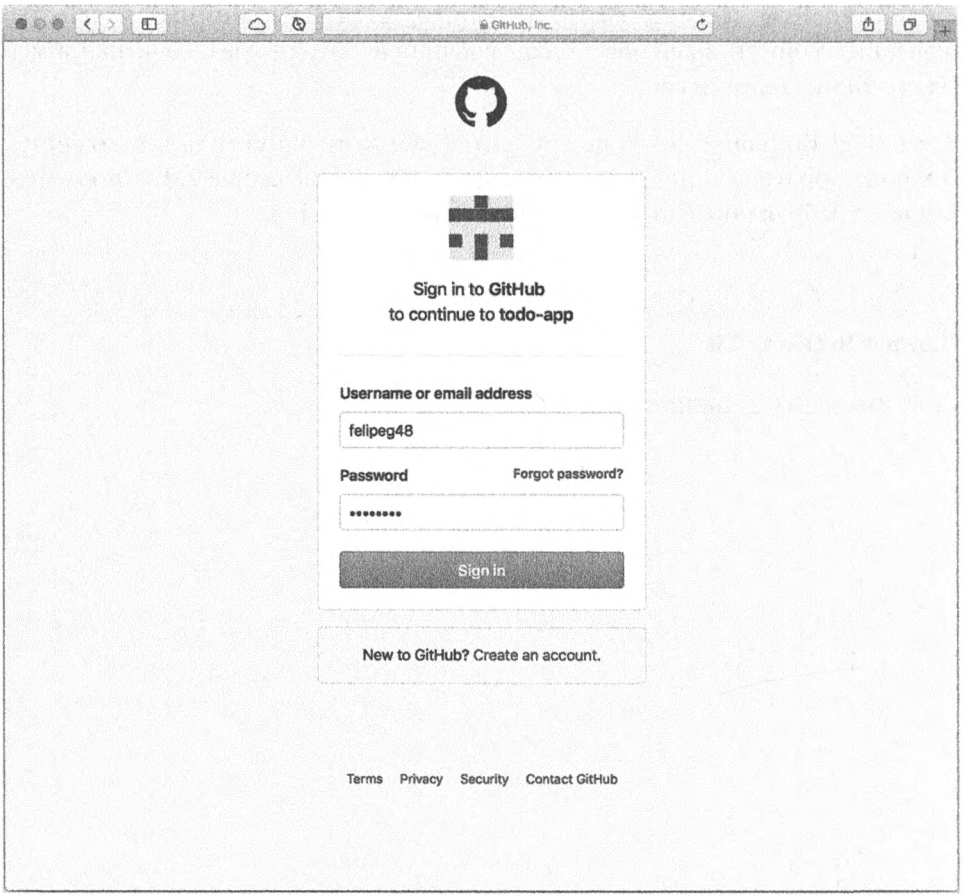


Рис. 8.12. Аутентификация GitHub

Поздравляю! Теперь вы знаете, насколько легко интегрировать OAuth2 с различными поставщиками идентификационной информации с помощью Spring Boot и Spring Security.

ПРИМЕЧАНИЕ

Программное решение для этого раздела можно найти в прилагаемом к книге исходном коде на веб-сайте Apress или в GitHub по адресу https://github.com/Apress/pro-spring-boot-2 либо в моем личном репозитории по адресу https://github.com/felipeg48/pro-spring-boot-2nd.

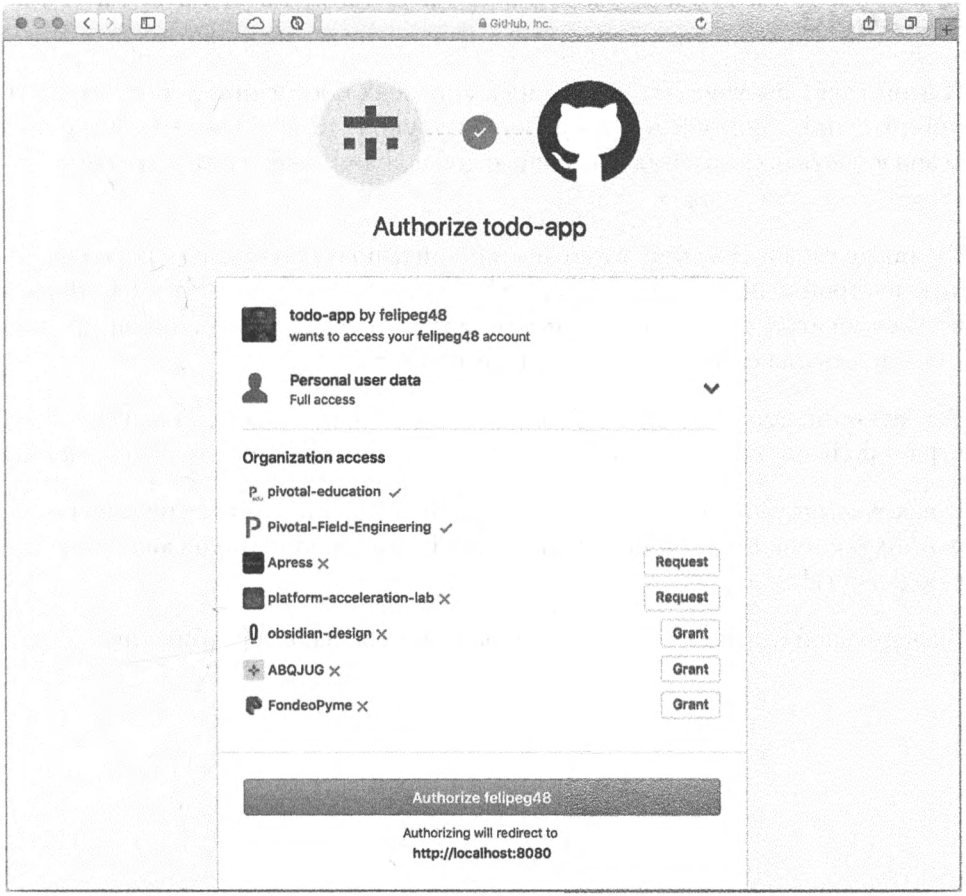


Рис. 8.13. Процесс авторизации GitHub

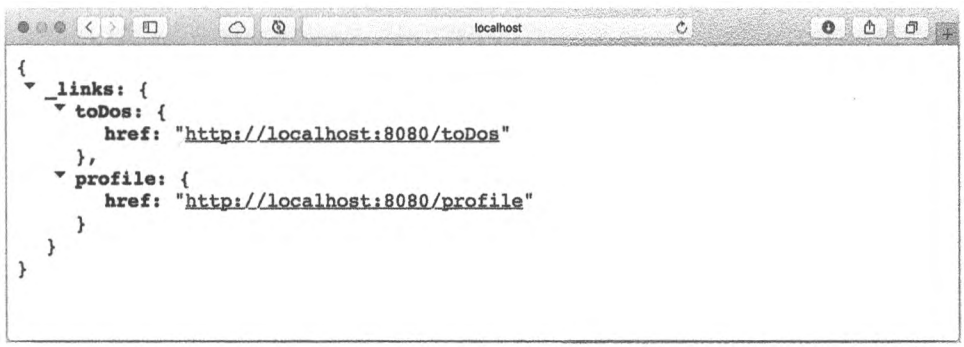


Рис. 8.14. По завершении процесса авторизации GitHub

Резюме

В этой главе вы узнали о различных способах обеспечивать безопасность приложений с помощью Spring Boot. Вы увидели, как просто обеспечить безопасность приложения с помощью добавления зависимости `spring-boot-security-starter`.

Вы также узнали, как просто можно изменить под свои нужды и переопределить настройки по умолчанию Spring Boot с помощью Spring Security. Можно воспользоваться свойствами `spring.security.*` или выполнить конфигурацию с помощью класса `WebSecurityConfigurerAdapter`.

Вы научились применять JDBC и соединять два приложения, одно из которых играет роль поставщика безопасности для аутентификации и авторизации.

И наконец, вы узнали, насколько просто использовать OAuth2 с помощью сторонних поставщиков идентификационной информации, таких как Facebook, Google, GitHub и др.

В следующей главе мы приступим к работе с брокерами сообщений.

9 Обмен сообщениями

Эта глава полностью посвящена обмену сообщениями. В ней на многочисленных примерах объясняется, как реализовать сервис сообщений Java (Java Message Service, JMS) с помощью ActiveMQ, расширенный протокол организации очереди сообщений (Advanced Message Queuing Protocol, AMQP) с помощью RabbitMQ, обмен сообщениями по типу публикация/подписка с помощью Redis и «простой/потоковый протокол обмена текстовыми сообщениями» (Simple/Streaming Text-Oriented Message Protocol, STOMP) с помощью Spring Boot.

Что такое обмен сообщениями

Обмен сообщениями — повсеместно встречающийся способ взаимодействия двух или более сущностей.

Обмен сообщениями между компьютерами — метод связи между аппаратными и программными компонентами или приложениями. В нем всегда участвует отправитель и один или несколько получателей. Обмен сообщениями может быть синхронным или асинхронным, по типу «публикация/подписка» или одноранговый, RPC, корпоративный, с помощью брокера сообщений, корпоративной шины сервисов (enterprise service bus, ESB), ориентированный на обработку сообщений программного обеспечения промежуточного уровня (message-oriented middleware, MOM) и т. д.

Благодаря обмену сообщениями становится возможной слабо связанная распределенная связь, в том смысле что получатель получает сообщение без

уведомления отправителя, вне зависимости от того, какое сообщение и как тот отправляет.

Конечно, об обмене сообщениями можно рассказывать очень много — начиная от старых испытанных методик и технологий до новых протоколов и паттернов обмена сообщениями, но задача данной главы — показать на примерах, как производит обмен сообщениями Spring Boot.

С учетом этого приступим к созданию примеров на основе некоторых из вышеупомянутых технологий и брокеров сообщений.

Использование JMS со Spring Boot

Начнем с использования JMS. Хотя это достаточно старая технология, она все еще применяется компаниями для унаследованных приложений. Компания Sun Microsystems создала JMS, чтобы обеспечить возможность синхронной и асинхронной передачи сообщений; он описывает, какие интерфейсы должны реализовывать брокеры сообщений, такие как WebLogic, IBM MQ, ActiveMQ, HornetQ и т. д.

JMS — технология, ориентированная исключительно на Java, хотя предпринимались и попытки сопряжения JMS с другими языками программирования; но сочетание различных технологий все еще остается весьма непростой или дорогостоящей задачей. Наверное, вы скажете, что это не так, ведь для интеграции JMS можно использовать Spring Integration, Google Protobuffers, Apache Thrift и другие технологии, но это все равно требует немалых усилий, поскольку приходится знать и сопровождать код для всех указанных технологий.

Создание приложения ToDo с использованием JMS

Начнем с создания приложения ToDo с использованием JMS со Spring Boot. Основная идея состоит в отправке запланированных дел брокеру JMS, получении их и сохранении.

Команда создателей Spring Boot предоставила несколько стартовых POM-файлов; в данном случае мы воспользуемся ActiveMQ — асинхронным брокером сообщений с открытым исходным кодом от Apache Foundation

(http://activemq.apache.org). Одно из главных его преимуществ — возможность задействования либо брокера в оперативной памяти, либо удаленного брокера (можете скачать и установить его, если хотите; в коде из этого раздела используется брокер в оперативной памяти, но я расскажу вам и как настроить удаленный брокер).

Откройте браузер и перейдите на уже известный вам сайт Spring Initializr (https://start.spring.io). Внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-jms`.
- Name (Название): `todo-jms`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `JMS (ActiveMQ)`, `Web`, `Lombok`, `JPA`, `REST Repositories`, `H2`, `MySQL`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в вашу любимую IDE (рис. 9.1).

Как вы видите из зависимостей, мы переиспользуем код JPA и REST Repositories из предыдущих глав. Вместо текстовых сообщений (распространенный подход к тестированию обмена сообщениями) мы будем использовать экземпляр класса `ToDo`, преобразуемый в формат JSON. Для этого вам придется вручную добавить следующую зависимость в свой файл `pom.xml` или `build.gradle`.

Если вы используете Maven, добавьте в файл `pom.xml` следующую зависимость:

```
<dependency>
      <groupId>com.fasterxml.jackson.core</groupId>
      <artifactId>jackson-databind</artifactId>
</dependency>
```

Если вы используете Gradle, добавьте следующую зависимость в файл `build.gradle`:

```
compile("com.fasterxml.jackson.core:jackson-databind")
```

Эта зависимость обеспечивает все JAR-файлы Jackson, необходимые для сериализации экземпляра `ToDo` на основе JSON.

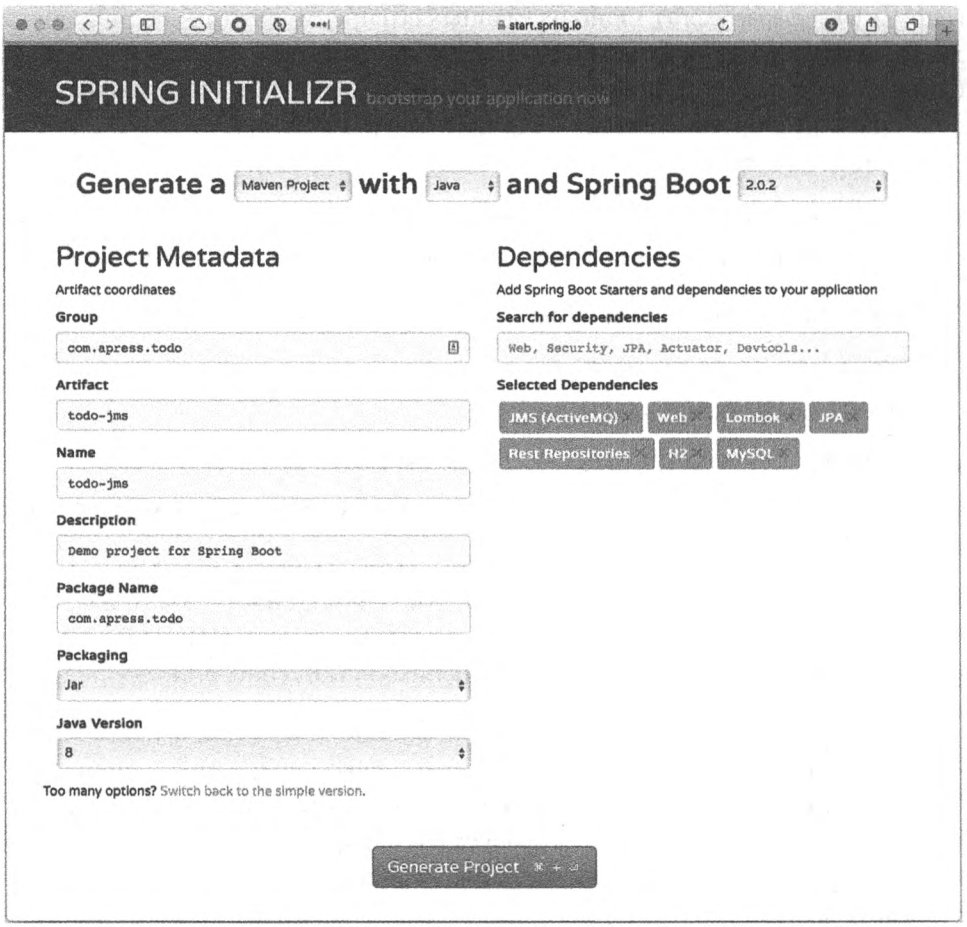


Рис. 9.1. Spring Initializr

В следующих разделах я покажу вам наиболее важные файлы, а также использование JMS в приложении ToDo. В этом примере применяется простой паттерн «Точка-точка», состоящий из *генератора данных* (producer), *очереди* (queue) и *потребителя данных* (consumer). Позднее я покажу, как настроить его для использования паттерна *«Издатель — подписчик»* (publisher — subscriber) с *генератором*, *темой* (топиком, topic) и несколькими *потребителями*.

Создайте класс `ToDoProducer`. Этот класс отвечает за отправку запланированного дела (объекта `ToDo`) в очередь JMS (листинг 9.1).

Листинг 9.1. com.apress.todo.jms.ToDoProducer.java

```
package com.apress.todo.jms;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class ToDoProducer {
     private static final Logger log =
         LoggerFactory.getLogger(ToDoProducer.class);

    private JmsTemplate jmsTemplate;

    public ToDoProducer(JmsTemplate jmsTemplate){
        this.jmsTemplate = jmsTemplate;
    }

    public void sendTo(String destination, ToDo toDo) {
        this.jmsTemplate.convertAndSend(destination, toDo);
        log.info("Producer> Message Sent");
    }
}
```

В листинге 9.1 приведен класс генератора. Этот класс регистрируется как компонент Spring в контексте приложения Spring, поскольку снабжен аннотацией `@Component`. В этом коде используется класс `JmsTemplate`, очень похожий на другие классы `*Template`, служащие обертками для всего стереотипного кода применяемой технологии. Через конструктор класса внедряется экземпляр `JmsTemplate`, задействуемый для отправки сообщения с помощью метода `convertAndSend`. При этом отправляется объект `ToDo` (строка в формате JSON). `JmsTemplate` обладает механизмом для его сериализации и отправки в активную очередь ActiveMQ.

Потребитель ToDo

Далее создадим класс потребителя, прослушивающий на предмет входящих сообщений из очереди ActiveMQ (листинг 9.2).

Листинг 9.2. com.apress.todo.jms.ToDoConsumer.java

```
package com.apress.todo.jms;

import com.apress.todo.domain.ToDo;
```

```
import com.apress.todo.repository.ToDoRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

import javax.validation.Valid;

@Component
public class ToDoConsumer {

    private Logger log = LoggerFactory.getLogger(ToDoConsumer.class);

    private ToDoRepository repository;

    public ToDoConsumer(ToDoRepository repository){
        this.repository = repository;
    }

    @JmsListener(destination = "${todo.jms.destination}", containerFactory =
        "jmsFactory")
    public void processToDo(@Valid ToDo todo){
        log.info("Consumer> " + todo);
        log.info("ToDo created> " + this.repository.save(todo));
    }
}
```

В листинге 9.2 показан потребитель. В этом классе мы используем объект `ToDoRepository`, прослушивающий на предмет любых сообщений из очереди ActiveMQ. Не забудьте воспользоваться аннотацией `@JmsListener`, чтобы метод обрабатывал все поступающие из очереди сообщения; в данном случае корректно сформированные объекты `ToDo` (для проверки любого из полей модели предметной области можно использовать аннотацию `@Valid`). У аннотации `@JmsListener` есть два атрибута. Атрибут `destination` уточняет, к какой очереди/теме подключаться (атрибут `destination` вычисляет значение свойства `todo.jms.destination`, которое мы создадим/будем использовать в следующем разделе). Атрибут `containerFactory` создается в качестве части конфигурации.

Настройка приложения ToDo

Пришло время настроить приложение ToDo для отправки и получения объектов `ToDo`. В листингах 9.1 и 9.2 приведены классы генератора и потребителя соответственно. В обоих классах используется экземпляр `ToDo`, так что не обойтись без сериализации. Большинство фреймворков сериализации Java требуют, чтобы ваши классы реализовали `java.io.Serializable`. Это простой

способ преобразования классов в байтовый вид, вызывающий тем не менее многолетние споры, ведь реализация `Serializable` снижает гибкость класса в смысле возможностей модификации его реализации после выпуска.

Фреймворк Spring предоставляет другой способ сериализации без реализации `Serializable` — с помощью интерфейса `MessageConverter`. В этом интерфейсе есть методы `toMessage` и `fromMessage`, в которые можно подключить подходящую технологию преобразования объектов.

Создадим конфигурацию, в которой для генератора и потребителя используются объекты `ToDo` (листинг 9.3).

Листинг 9.3. com.apress.todo.config.ToDoConfig.java

```
package com.apress.todo.config;
import com.apress.todo.error.ToDoErrorHandler;
import com.apress.todo.validator.ToDoValidator;
import org.springframework.boot.autoconfigure.jms.
  DefaultJmsListenerContainerFactoryConfigurer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.annotation.JmsListenerConfigurer;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;
import org.springframework.jms.config.JmsListenerContainerFactory;
import org.springframework.jms.config.JmsListenerEndpointRegistrar;
import org.springframework.jms.support.converter.
  MappingJackson2MessageConverter;
import org.springframework.jms.support.converter.MessageConverter;
import org.springframework.jms.support.converter.MessageType;
import org.springframework.messaging.handler.annotation.support.
  DefaultMessageHandlerMethodFactory;

import javax.jms.ConnectionFactory;

@Configuration
public class ToDoConfig {
    @Bean
    public MessageConverter jacksonJmsMessageConverter() {
        MappingJackson2MessageConverter converter = new
             MappingJackson2MessageConverter();
        converter.setTargetType(MessageType.TEXT);
        converter.setTypeIdPropertyName("_class_");
        return converter;
    }

    @Bean
     public JmsListenerContainerFactory<?> jmsFactory(ConnectionFactory
  connectionFactory, DefaultJmsListenerContainerFactoryConfigurer configurer) {
```

```
        DefaultJmsListenerContainerFactory factory = new
        DefaultJmsListenerContainerFactory();
        factory.setErrorHandler(new ToDoErrorHandler());
        configurer.configure(factory, connectionFactory);
        return factory;
    }

    @Configuration
    static class MethodListenerConfig implements JmsListenerConfigurer{

        @Override
        public void configureJmsListeners (JmsListenerEndpointRegistrar
            jmsListenerEndpointRegistrar){
            jmsListenerEndpointRegistrar.setMessageHandlerMethodFactory
                (myHandlerMethodFactory());
        }

        @Bean
        public DefaultMessageHandlerMethodFactory myHandlerMethodFactory ()
{
            DefaultMessageHandlerMethodFactory factory = new
                 DefaultMessageHandlerMethodFactory();
            factory.setValidator(new ToDoValidator());
            return factory;
        }
    }
}
```

В листинге 9.3 приведен используемый в нашем приложении класс `ToDoConfig`. Проанализируем его.

- Аннотация `@Configuration`. Эта уже известная вам аннотация указывает контейнеру Spring, что соответствующий класс служит для конфигурации контекста приложения Spring.
- `MessageConverter`. Метод `jacksonJmsMessageConverter` возвращает объект интерфейса `MessageConverter`. Этот интерфейс продвигает создание реализаций методов `toMessage` и `fromMessage`, упрощающих подключение любых нужных сериализаций/преобразований. В данном случае речь идет о преобразовании в/из JSON с помощью реализации класса `MappingJackson2MessageConverter`. Этот класс – одна из предоставляемых фреймворком Spring реализаций по умолчанию. Она использует библиотеки Jackson, которые, в свою очередь, используют функции отображения для преобразования из/в JSON в объект/из объекта. А поскольку мы работаем с экземплярами класса `ToDo`, необходимо указать целевой

тип (`setTargetType`), то есть JSON-объект получается как текст, а имя свойства `type-id` (`setTypeIdPropertyName`) однозначно идентифицирует получаемое из генератора и потребителя свойство. Название содержащего идентификатор типа свойства должно всегда соответствовать как генератору, так и потребителю. Им может служить любое подходящее значение (желательно нечто, что вы сможете узнать, поскольку оно используется в названии (включая пакет) преобразуемого в JSON-класс/из него), другими словами, `com.apress.todo.domain`. Класс `ToDo` должен быть доступен как генератору, так и потребителю, чтобы функция отображения знала, откуда/куда отображать класс.

- Интерфейс `JmsListenerContainerFactory`. Метод `jmsFactory` возвращает объект интерфейса `JmsListenerContainerFactory`. Для этого компонента нужны компоненты `ConnectionFactory` и `DefaultJmsListenerContainerFactoryConfigurer` (их оба внедряет Spring), он создает объект `DefaultJmsListenerContainerFactory`, который задает обработчик ошибок. Этот компонент используется в аннотации `@JmsListener`, для чего задается значение атрибута `containerFactory`.
- Функциональный интерфейс `JmsListenerConfigurer`. В этом интерфейсе мы создаем статическую конфигурацию. Его реализует класс `MethodListenerConfig`. Этот интерфейс требует регистрации компонента с конфигурацией средства проверки корректности данных (класса `ToDoValidator`), в этом случае компонента `DefaultMessageHandlerMethodFactory`.

Если вы пока что не хотите выполнять проверку корректности данных, можете удалить класс `MethodListenerConfig` и вызов метода `setErrorHandler` из объявления компонента `jmsFactory`; но если вам было бы интересно поэкспериментировать с проверкой корректности данных, то необходимо создать класс `ToDoValidator` (листинг 9.4).

Листинг 9.4. com.apress.todo.validator.ToDoValidator.java

```
package com.apress.todo.validator;

import com.apress.todo.domain.ToDo;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

public class ToDoValidator implements Validator {
    @Override
```

```
    public boolean supports(Class<?> clazz) {
        return clazz.isAssignableFrom(ToDo.class);
    }

    @Override
    public void validate(Object target, Errors errors) {
        ToDo toDo = (ToDo)target;

        if (toDo == null) {
            errors.reject(null, "ToDo cannot be null");
        }else {
             if (toDo.getDescription() == null ||
                             toDo.getDescription().isEmpty())
                 errors.rejectValue("description",null,
                   "description cannot be null or empty");
        }
    }
}
```

В листинге 9.4 приведен класс для проверки корректности данных, вызываемый для каждого сообщения и проверяющий, не пусто ли (или равно `null`) поле `description`. Этот класс реализует интерфейс `Validator`, в том числе методы `supports` и `validate`.

Вот код класса `ToDoErrorHandler`:

```
package com.apress.todo.error;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.util.ErrorHandler;

public class ToDoErrorHandler implements ErrorHandler {
    private static Logger log = LoggerFactory.getLogger(ToDoErrorHandler.class);

    @Override
    public void handleError(Throwable t) {
        log.warn("ToDo error...");
        log.error(t.getCause().getMessage());
    }
}
```

Как вы видите, этот класс реализует интерфейс `ErrorHandler`.

Теперь создадим класс `ToDoProperties` для хранения свойства `todo.jms.destination`, указывающего, к какой очереди/теме необходимо подключаться (листинг 9.5).

Листинг 9.5. com.apress.todo.config.ToDoProperties.java

```
package com.apress.todo.config;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

@Data
@ConfigurationProperties(prefix = "todo.jms")
public class ToDoProperties {

    private String destination;
}
```

В листинге 9.5 приведен класс `ToDoProperties`. Как вы помните, в листинге 9.2 (класс `ToDoConsumer`) метод `processToDo` был снабжен аннотацией `@JmsListener` с атрибутом `destination`. Этот атрибут получает значение путем вычисления описанного внутри класса выражения `${todo.jms.destination}` *SpEL* (языка выражений Spring).

Задать значение этого свойства можно в файле `application.properties`:

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop

# ToDo JMS
todo.jms.destination=toDoDestination
```

Запуск приложения ToDo

Теперь создадим класс конфигурации для отправки сообщения в очередь с помощью генератора (листинг 9.6).

Листинг 9.6. com.apress.todo.config.ToDoSender.java

```
package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import com.apress.todo.jms.ToDoProducer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ToDoSender {
```

```
    @Bean
    public CommandLineRunner sendToDos(@Value("${todo.jms.destination}")
        String destination, ToDoProducer producer){
        return args -> {
            producer.sendTo(destination,new ToDo("workout tomorrow morning!"));
        };
    }

}
```

В листинге 9.6 приведен класс конфигурации, отправляющий сообщение с помощью экземпляра `ToDoProducer` с пунктом назначения, указанным в свойстве `todo.jms.destination`.

Для запуска приложения можно использовать либо IDE (если вы его импортировали туда), либо обертку Maven:

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

Либо обертку Gradle:

```
./gradlew bootRun
```

В журнале вы должны при этом увидеть следующий текст:

```
Producer> Message Sent
Consumer> ToDo(id=null, description=workout tomorrow morning!,
 created=null, modified=null, completed=false)
ToDo created> ToDo(id=8a808087645bd67001645bd6785b0000, description=workout
 tomorrow morning!, created=2018-07-02T10:32:19.546, modified=2018-07-
 02T10:32:19.547, completed=false)
```

Можете взглянуть на конечную точку http://localhost:8080/toDos и убедиться, что объект `ToDo` был действительно создан.

Использование паттерна публикации/подписки JMS

А если вы хотите использовать паттерн публикации/подписки, при котором сообщение получает несколько потребителей (благодаря подписке на тему), то сейчас я расскажу вам, что надо делать.

Применение Spring Boot упрощает настройку паттерна публикации/подписки. При использовании прослушивателя по умолчанию (`@JmsListener(destination)` — контейнер прослушивателя по умолчанию) мож-

но воспользоваться свойством `spring.jms.pub-sub-domain=true` в файле `application.properties`.

Но пользовательский контейнер прослушивателя можно настроить программным образом.

```
@Bean
public DefaultMessageListenerContainer jmsListenerContainerFactory() {
    DefaultMessageListenerContainer dmlc = new
    DefaultMessageListenerContainer();
    dmlc.setPubSubDomain(true);
    // Прочие настройки ...
    return dmlc;
}
```

Удаленный сервер ActiveMQ

Приложение ToDo использует размещаемый в оперативной памяти брокер (`spring.activemq.in-memory=true`). Возможно, это хорошо подходит для демонстрации или тестирования, но на практике применяется удаленный сервер ActiveMQ, для работы с которым необходимо добавить следующие ключи в файл `application.properties` (отредактируйте его соответствующим образом).

```
spring.activemq.broker-url=tcp://my-awesome-server.com:61616
spring.activemq.user=admin
spring.activemq.password=admin
```

Для настройки брокера ActiveMQ можно использовать много других свойств. Список ключей `spring.activemq.*` вы можете найти на странице https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html.

Использование RabbitMQ со Spring Boot

Протоколы, начиная с первых попыток реализации JMS такими компаниями, как Sun, Oracle и IBM, а также Microsoft с их MSMQ, были проприетарными. В JMS описан API взаимодействия, но смешение технологий или языков программирования только доставляет лишнюю головную боль разработчику. Благодаря команде разработчиков холдинга JPMorgan был создан расширенный протокол организации очереди сообщений (Advanced Message Queuing Protocol, AMQP), представляющий собой прикладной слой с открытым стандартом для MOM. Другими словами, AMQP — протокол

передачи данных в смысле возможности использования с ним любого языка программирования или технологии.

Брокеры сообщений непрерывно соревнуются друг с другом в ошибкоустойчивости, надежности и масштабируемости, но важнее всего — их быстродействие. Я работал со множеством брокеров, самым простым в использовании и масштабировании и наиболее быстродействующим был RabbitMQ, реализующий протокол AMQP.

Описание всех деталей и понятий RabbitMQ потребовало бы отдельной книги, но я попытаюсь пояснить некоторые из них на примере из данного раздела.

Установка RabbitMQ

Прежде чем говорить о RabbitMQ, следует его установить. Если вы работаете в Mac OS X/Linux, можете воспользоваться командой `brew`.

```
$ brew upgrade
$ brew install rabbitmq
```

Если же вы работаете в Unix или Windows-системе, то можете зайти на веб-сайт RabbitMQ и воспользоваться одним из установочных пакетов (www.rabbitmq.com/download.html). RabbitMQ написан на языке Erlang, так что главное, что для него нужно, — наличие среды выполнения Erlang в системе. В настоящее время все установочные пакеты RabbitMQ включают все зависимости Erlang[1]. Необходимо только убедиться, что путь к исполняемым файлам включен в системную переменную `PATH` (для Windows и Linux, в зависимости от используемой операционной системы). При использовании `brew` заботиться о переменной `PATH` не нужно.

RabbitMQ/AMQP: точки обмена, привязки и очереди

В AMQP описывается три довольно простых, но отличающихся от мира JMS понятия. Во-первых, это *точки обмена* (exchanges) — сущности, которым отправляются сообщения. Точка обмена получает сообщение и производит его маршрутизацию нулю или более *очередей* (queues). При этой маршрутизации используется алгоритм, учитывающий тип точки обмена и правила, которые называются *привязками* (bindings).

[1] На самом деле для RabbitMQ требуется установить 64-битную версию Erlang для Windows.

Протокол AMQP определяет пять[1] типов точек обмена: *по умолчанию* (default), *прямые* (direct), *с разветвлением* (fanout), *тематические* (topic) и *заголовочные* (headers). Эти типы точек обмена показаны на рис. 9.2.

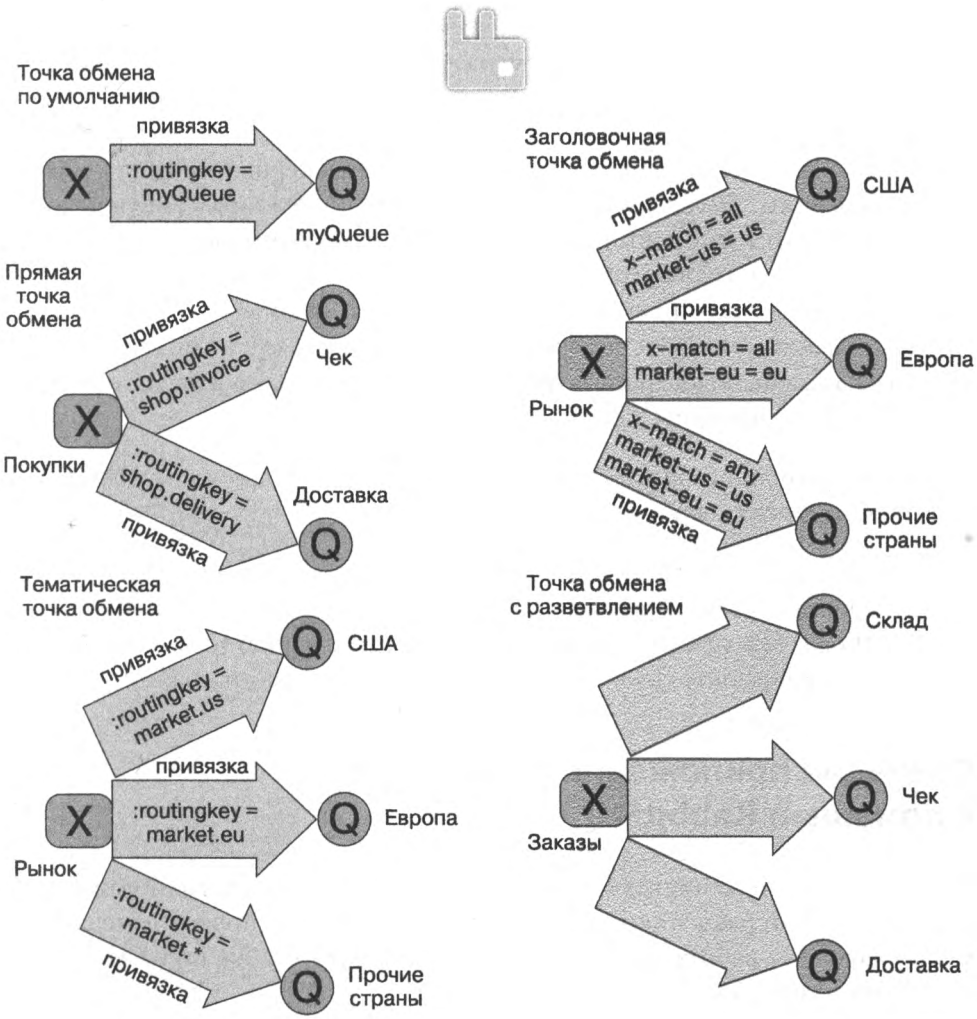


Рис. 9.2. Точки обмена/привязки/очереди AMQP

[1] На самом деле в различных версиях спецификаций AMQP список типов точек обмена различен. Помимо упомянутых в тексте типов, выделяют еще системные точки обмена. См., например, подраздел 3.1.3 на странице https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf.

На рис. 9.2 приведены возможные типы точек обмена. Основная идея состоит в отправке сообщения в точку обмена, включая ключ маршрутизации, после чего точка обмена, в зависимости от типа, отправляет сообщение в очередь (или не отправляет, в случае неподходящего ключа маршрутизации).

Точки обмена *по умолчанию* автоматически связываются с каждой из создаваемых очередей. *Прямые* точки обмена связываются с очередями по ключу маршрутизации; этот тип обмена можно считать взаимнооднозначной привязкой. *Тематические* точки обмена похожи на прямые; единственное отличие состоит в возможности указать в привязке джокерный символ в ключе маршрутизации. *Заголовочные* точки обмена аналогичны тематическим; единственное отличие состоит в привязке на основе заголовков сообщений (возможности таких точек обмена чрезвычайно широки, для сопоставления с заголовками можно использовать выражения `all` и `any`). Точки обмена *с разветвлением* копируют сообщение во все связанные с ними очереди; такую точку обмена можно рассматривать как средство «широковещательной» трансляции сообщений.

Больше информации обо всем этом можно найти на странице www.rabbitmq.com/tutorials/amqp-concepts.html.

В примере из этого раздела используется точка обмена по умолчанию, то есть ключ маршрутизации равен названию очереди. При всяком создании очереди RabbitMQ создает привязку точки обмена по умолчанию (фактическое название которой равно пустой строке) к очереди на основе названия очереди.

Создание приложения ToDo с помощью RabbitMQ

Вернемся к приложению ToDo и добавим в него обмен сообщениями на основе AMQP. По сравнению с предыдущим приложением ничего не меняется, мы продолжаем работать с экземплярами `ToDo`, отправляя/получая JSON-сообщения с преобразованием их в объекты.

Для начала откройте браузер и перейдите на уже известный вам сайт Spring Initializr. Добавьте следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-rabbitmq`.

- Name (Название): `todo-rabbitmq`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `RabbitMQ`, `Web`, `Lombok`, `JPA`, `REST Repositories`, `H2`, `MySQL`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в вашу любимую IDE (рис. 9.3).

Можете скопировать код проекта JPA/REST из предыдущих глав.

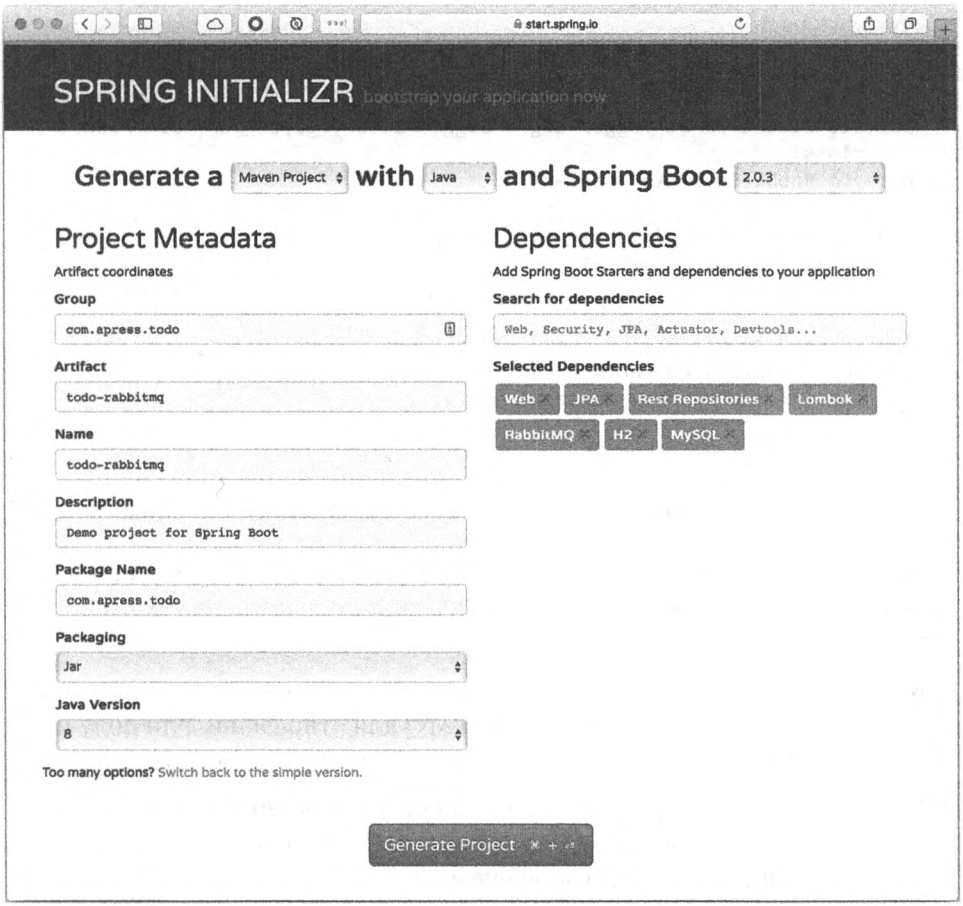


Рис. 9.3. Spring Initializr: https://start.spring.io

Генератор ToDo

Начнем с создания класса генератора, который будет отправлять сообщения в точку обмена (точка обмена по умолчанию — прямая) (листинг 9.7).

Листинг 9.7. com.apress.todo.rmq.ToDoProducer.java

```
package com.apress.todo.rmq;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.stereotype.Component;

@Component
public class ToDoProducer {

    private static final Logger log = LoggerFactory.getLogger(ToDoProducer.
        class);
    private RabbitTemplate template;

    public ToDoProducer(RabbitTemplate template){
        this.template = template;
    }

    public void sendTo(String queue, ToDo toDo){
        this.template.convertAndSend(queue,toDo);
        log.info("Producer> Message Sent");
    }
}
```

В листинге 9.7 приведен класс `ToDoProducer.java`. Изучим его внимательнее.

- Аннотация `@Component`. Эта аннотация отмечает класс как кандидат в компоненты для последующего автоматического обнаружения контейнером Spring.
- Класс `RabbitTemplate`. Это вспомогательный класс, упрощающий синхронное/асинхронное обращение к RabbitMQ для отправки и/или получения сообщений. Очень похоже на уже знакомый вам класс `JmsTemplate`.
- Метод `sendTo(routingKey, message)`. В списке параметров этого метода — ключ маршрутизации и само сообщение. В данном случае роль ключа маршрутизации играет название очереди. В этом методе используется экземпляр `rabbitTemplate` для вызова метода `convertAndSend`, принимающего

в качестве параметров ключ маршрутизации и сообщение. Напомню, что сообщение отправляется в точку обмена (по умолчанию), которая производит его маршрутизацию в нужную очередь. Ключ маршрутизации совпадает с названием очереди. Учтите также, что по умолчанию RabbitMQ всегда связывает точку обмена по умолчанию (прямую точку обмена) с очередью, а ключом маршрутизации служит название этой очереди.

Потребитель ToDo

Пришло время создать класс потребителя — прослушиватель для заданной очереди (листинг 9.8).

Листинг 9.8. com.apress.todo.rmq.ToDoConsumer.java

```
package com.apress.todo.rmq;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.stereotype.Component;

@Component
public class ToDoConsumer {

    private Logger log = LoggerFactory.getLogger(ToDoConsumer.class);
    private ToDoRepository repository;

    public ToDoConsumer(ToDoRepository repository){
        this.repository = repository;
    }

    @RabbitListener(queues = "${todo.amqp.queue}")
    public void processToDo(ToDo todo){
        log.info("Consumer> " + todo);
        log.info("ToDo created> " + this.repository.save(todo));
    }
}
```

В листинге 9.8 приведен класс `ToDoConsumer.java`. Изучим его внимательнее.

- Уже знакомая вам аннотация `@Component`. Она отмечает класс как кандидат в компоненты для последующего автоматического обнаружения контейнером Spring.

- Аннотация `@RabbitListener`. Помечает метод (ее можно использовать и на уровне класса) для создания обработчика входящих сообщений в том смысле, что создает прослушиватель, подключенный к очереди RabbitMQ и передающий эти сообщения данному методу. «За кулисами» прослушиватель делает все возможное для преобразования сообщения в подходящий тип с помощью соответствующего средства преобразования сообщений (реализации интерфейса `org.springframework.amqp.support.converter.MessageConverter`, входящего в состав проекта `spring-amqp`); в данном случае он преобразует сообщение из формата JSON в экземпляр `ToDo`.

Как вы видите из классов `ToDoProducer` и `ToDoConsumer`, код очень прост. Для создания аналогичной функциональности с помощью одного только Java-клиента RabbitMQ (www.rabbitmq.com/java-client.html) как минимум понадобилось бы больше строк кода для создания соединения, канала и отправки сообщения; а при написании потребителя пришлось бы открыть соединение, создать канал, создать простейший потребитель и обрабатывать в цикле все входящие сообщения. Это немало работы для простого потребителя или генератора. Именно поэтому команда создателей Spring AMQP разработала подобный способ решения сложной задачи всего в нескольких строках кода.

Конфигурация приложения ToDo

Теперь зададим конфигурацию нашего приложения. Напомню, что речь идет об отправке экземпляров `ToDo`, так что фактически нужна та же самая конфигурация, что и для JMS. Необходимо задать средство преобразования и контейнер прослушивателя (листинг 9.9).

Листинг 9.9. com.apress.todo.config.ToDoConfig.java

```
package com.apress.todo.config;

import org.springframework.amqp.core.Queue;
import org.springframework.amqp.rabbit.config.
  SimpleRabbitListenerContainerFactory;
import org.springframework.amqp.rabbit.connection.ConnectionFactory;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.Jackson2JsonMessageConverter;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ToDoConfig {
```

```
    @Bean
     public SimpleRabbitListenerContainerFactory
        rabbitListenerContainerFactory (ConnectionFactory connectionFactory) {
        SimpleRabbitListenerContainerFactory factory = new
        SimpleRabbitListenerContainerFactory();
        factory.setConnectionFactory(connectionFactory);
        factory.setMessageConverter(new Jackson2JsonMessageConverter());
        return factory;
    }

    @Bean
    public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory){
        RabbitTemplate template = new RabbitTemplate(connectionFactory);
        template.setMessageConverter(new Jackson2JsonMessageConverter());
        return template;
    }

    @Bean
    public Queue queueCreation(@Value("${todo.amqp.queue}") String queue){
        return new Queue(queue,true,false,false);
    }
}
```

В листинге 9.9 приведена конфигурация. В ней описано несколько компонентов, взглянем на них внимательнее.

- Класс `SimpleRabbitListenerContainerFactory`. Эта фабрика требуется при использовании аннотации `@RabbitListener` для пользовательских настроек, поскольку мы работаем с экземплярами `ToDo`; необходимо обязательно задать средство преобразования.
- Класс `Jackson2JsonMessageConverter`. Этот преобразователь применяется для генерации (с помощью `RabbitTemplate`) и для потребления (`@RabbitListener`); для отображения и преобразования он использует библиотеки Jackson.
- Класс `RabbitTemplate`. Вспомогательный класс, умеющий отправлять и получать сообщения. В данном случае необходимо настроить его для генерации объектов JSON с помощью преобразователя Jackson.
- `Queue`. Можно создать очередь вручную, но в данном случае мы создаем ее программным образом: передаем название очереди, указываем, должна ли она быть устойчивой к перезагрузке сервера, применяться только данным соединением и автоматически уничтожаться в случае, если больше не используется.

Напоминаю, что в протоколе AMQP точка обмена должна быть связана с очередью, так что в этом конкретном примере во время выполнения создается

очередь spring-boot и изначально все очереди привязаны к точке обмена по умолчанию. Именно поэтому мы не указываем никакой информации о точке обмена. Так, генератор, отправляет сообщение сначала в точку обмена по умолчанию, а затем оно маршрутизируется в очередь (spring-boot).

Запуск приложения ToDo

Создадим класс, который будет отправлять сообщения ToDo (листинг 9.10).

Листинг 9.10. com.apress.todo.config.ToDoSender.java

```
package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import com.apress.todo.rmq.ToDoProducer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class ToDoSender {

    @Bean
     public CommandLineRunner sendToDos(@Value("${todo.amqp.queue}") String
        destination, ToDoProducer producer){
        return args -> {
             producer.sendTo(destination,new ToDo("workout tomorrow
                 morning!"));
        };
    }
}
```

Добавьте следующие ключи (для объявления очереди, в которую будет производиться отправка/из которой будут потребляться сообщения) в файл application.properties.

Прежде чем запустить пример, убедитесь, что ваш сервер RabbitMQ запущен. Запустить его можно с помощью выполнения в терминале следующей команды.

```
$ rabbitmq-server
```

Проверьте доступ к веб-консоли RabbitMQ, перейдя по адресу http://localhost:15672/ с учетными данными guest/guest. В случае проблем с досту-

пом к веб-консоли проверьте, запущен ли плагин управления RabbitMQ (`rabbitmq_management`), с помощью следующей команды:

```
$ rabbitmq-plugins list
```

Если флажки во всем списке сняты, значит, плагин управления не запущен (обычное дело при только что установленном RabbitMQ). Для активации этого плагина выполните следующую команду:

```
$ rabbitmq-plugins enable rabbitmq_management --online
```

Теперь можете попробовать снова. В результате вы должны увидеть примерно такую веб-консоль, как на рис. 9.4.

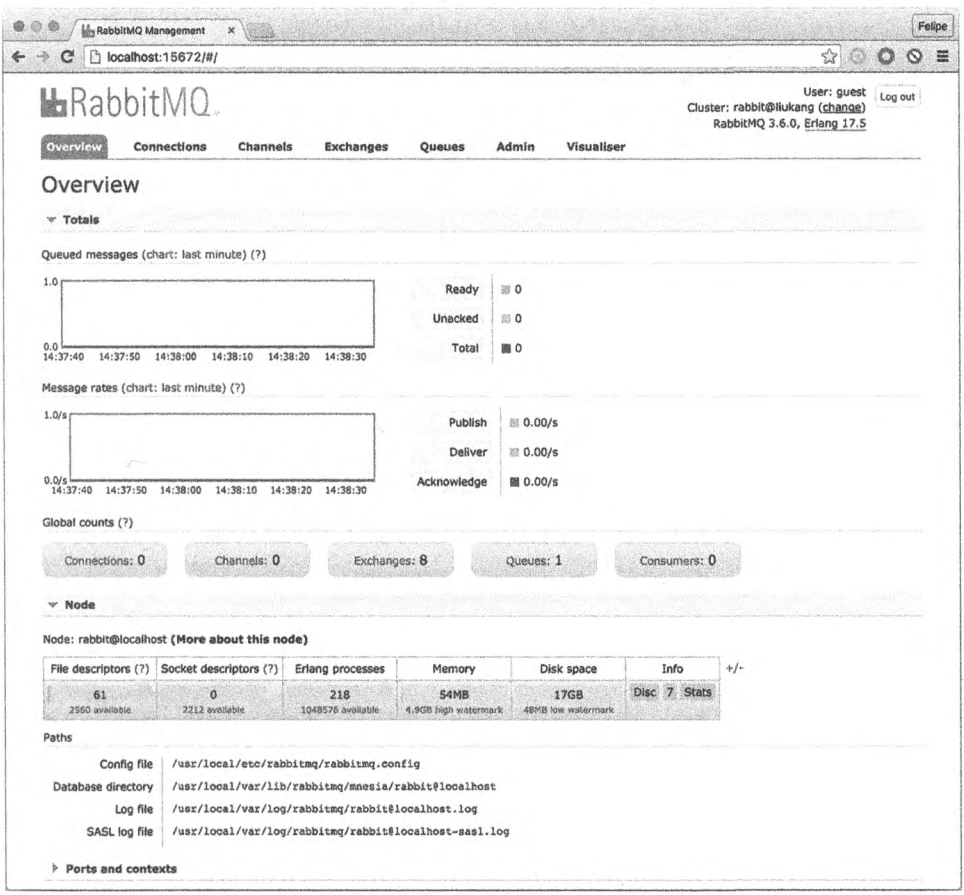


Рис. 9.4. Веб-консоль управления RabbitMQ

На рис. 9.4 показана веб-консоль RabbitMQ. Теперь вы можете запустить проект обычным образом, с помощью вашей IDE. Если вы используете Maven, выполните:

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

Если вы применяете Gradle, выполните:

```
$./gradlew bootRun
```

После выполнения этой команды вы должны увидеть примерно следующий вывод:

```
Producer> Message Sent
Consumer> ToDo(id=null, description=workout tomorrow morning!,
 created=null, modified=null, completed=false)
ToDo created> ToDo(id=8a808087645bd67001645bd6785b0000,
 description=workout tomorrow morning!, created=2018-07-02T10:32:19.546,
 modified=2018-07-02T10:32:19.547, completed=false)
```

Если взглянуть теперь на закладку Queues в веб-консоли RabbitMQ, вы должны увидеть там очередь `spring-boot` (рис. 9.5).

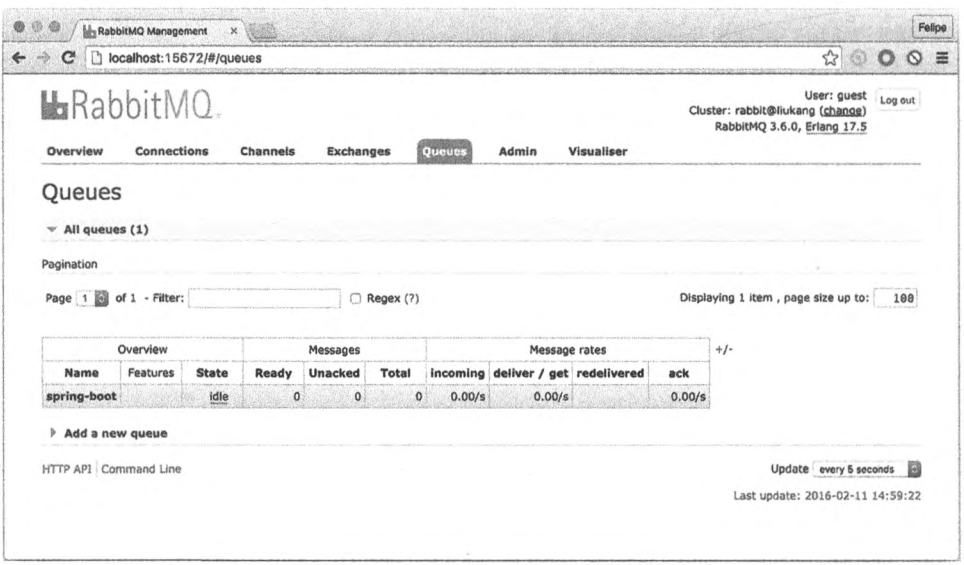


Рис. 9.5. Закладка Queues веб-консоли RabbitMQ

На рис. 9.5 показана закладка Queues из веб-консоли RabbitMQ. Отправленное вами сообщение было сразу же доставлено. Если хотите поэкспери-

ментировать еще немного, можете модифицировать класс ToDoSender так, как показано в листинге 9.11, не забудьте только остановить приложение.

Листинг 9.11. Версия 2 класса com.apress.todo.config.ToDoSender.java

```
package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import com.apress.todo.rmq.ToDoProducer;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Configuration;
import org.springframework.scheduling.annotation.EnableScheduling;
import org.springframework.scheduling.annotation.Scheduled;

import java.text.SimpleDateFormat;
import java.util.Date;

@EnableScheduling
@Configuration
public class ToDoSender {

    @Autowired
    private ToDoProducer producer;
    @Value("${todo.amqp.queue}")
    private String destination;
    private SimpleDateFormat dateFormat = new SimpleDateFormat("HH:mm:ss");

    @Scheduled(fixedRate = 500L)
    private void sendToDos(){
        producer.sendTo(destination,new ToDo("Thinking on Spring Boot at "
            + dateFormat.format(new Date())));
    }
}
```

В листинге 9.11 приведена модифицированная версия класса ToDoSender.java. Изучим эту новую версию.

- Аннотация @EnableScheduling. Эта аннотация указывает (посредством автоконфигурации) контейнеру Spring, что необходимо создать класс org.springframework.scheduling.annotation.ScheduledAnnotationBeanPostProcessor. Она регистрирует все снабженные аннотацией @Scheduled методы для вызова их реализацией интерфейса org.springframework.scheduling.TaskScheduler в соответствии с указанными в аннотации @Scheduled параметрами fixedRate, fixedDelay или выражением cron.
- @Scheduled(fixedDelay = 500L). Эта аннотация указывает интерфейсу TaskScheduler выполнять метод sendToDos через фиксированный отрезок времени 500 миллисекунд. Это значит, что каждые полсекунды в очередь будет отправляться сообщение.

Оставшаяся часть приложения уже вам знакома. Если снова запустить приложение, вы увидите бесконечную отправку сообщений. Взгляните во время его работы на консоль RabbitMQ. Можно также вставить цикл `for` для отправки большего числа сообщений за полсекунды.

Удаленный сервер RabbitMQ

Для доступа к удаленному серверу RabbitMQ необходимо добавить в файл `application.properties` следующие свойства[1]:

```
spring.rabbitmq.host=mydomain.com
spring.rabbitmq.username=rabbituser
spring.rabbitmq.password=thisissecured
spring.rabbitmq.port=5672
spring.rabbitmq.virtual-host=/production
```

Вы всегда можете узнать обо всех свойствах RabbitMQ в документации Spring Boot по адресу https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html.

Теперь вы знаете, насколько легко использовать RabbitMQ со Spring Boot. Если хотите узнать больше о RabbitMQ и технологии Spring AMQP, то можете получить дополнительную информацию на основной странице этого проекта: http://projects.spring.io/spring-amqp/.

Можете теперь остановить сервер RabbitMQ, нажав Ctrl+C в терминале, где запустили его. Существуют и другие возможности использования RabbitMQ, например создание кластера и обеспечение высокой доступности. Больше информации об этом можно найти на сайте www.rabbitmq.com.

Обмен сообщениями в Redis с помощью Spring Boot

Настала очередь Redis. Redis (REmote DIctionary Server) представляет собой базу данных NoSQL — хранилище пар «ключ/значение». Redis написана на языке C и, несмотря на небольшой объем используемой оперативной памяти, отличается высокой надежностью, масштабируемостью, быстродействием и большими возможностями. Основная ее задача — хранение структур данных,

[1] Разумеется, поменяв соответствующим образом их значения.

таких как списки, хеши, строковые значения, множества и отсортированные множества. Одна из основных ее возможностей — обмен сообщениями по типу «публикация/подписка». Поэтому мы и воспользуемся Redis в качестве брокера сообщений.

Установка Redis

Процедура установки Redis очень проста. Если вы используете Mac OS X/Linux, можете воспользоваться командой `brew` и выполнить следующее:

```
$ brew update && brew install redis
```

Если же вы работаете в Unix или Windows-системе, то можете зайти на вебсайт Redis и скачать один из установочных пакетов Redis[1] по адресу http://redis.io/download.

Создание приложения ToDo с использованием Redis

Использовать Redis для обмена сообщениями по типу «публикация/подписка» очень просто, никаких серьезных отличий от других технологий нет. Мы сейчас реализуем отправку и получение запланированных дел на основе паттерна публикации/подписки с помощью Redis.

Для начала откройте свой любимый браузер и перейдите на сайт Spring Initializr. Добавьте следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-redis`.
- Name (Название): `todo-redis`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Redis`, `Web`, `Lombok`, `JPA`, `REST Repositories`, `H2`, `MySQL`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в вашу любимую IDE (рис. 9.6).

[1] Уточнение: на этом сайте доступен только исходный код, который необходимо скомпилировать для получения исполняемых файлов. Впрочем, существуют и бинарные сборки, правда, для Windows они все довольно сильно отстают от текущей версии.

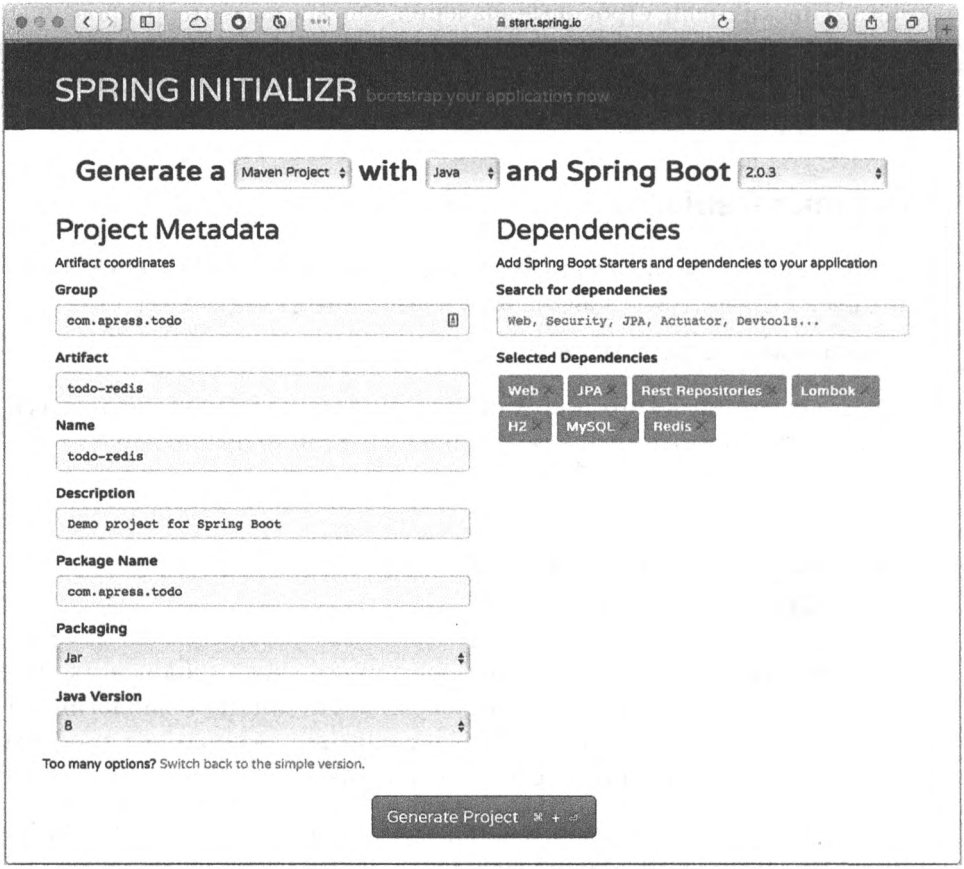


Рис. 9.6. Spring Initializr

Мы воспользуемся классом предметной области ToDo и репозиторием из предыдущих глав.

Генератор ToDo

Начнем с создания класса генератора, задача которого будет состоять в отправке экземпляра `ToDo` в конкретную тему (листинг 9.12).

Листинг 9.12. com.apress.todo.redis.ToDoProducer.java

```
package com.apress.todo.redis;

import com.apress.todo.domain.ToDo;
```

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.stereotype.Component;

@Component
public class ToDoProducer {

    private static final Logger log = LoggerFactory.getLogger(ToDoProducer.
        class);
    private RedisTemplate redisTemplate;

    public ToDoProducer(RedisTemplate redisTemplate){
        this.redisTemplate = redisTemplate;
    }

    public void sendTo(String topic, ToDo toDo){
        log.info("Producer> ToDo sent");
        this.redisTemplate.convertAndSend(topic, toDo);
    }
}
```

В листинге 9.12 приведен класс генератора. Он практически идентичен предыдущим технологиям. В нем используется класс паттерна `*Template`; в данном случае `RedisTemplate`, отправляющий экземпляры `ToDo` в конкретную тему.

Потребитель ToDo

Далее создадим потребитель для подписки на тему (листинг 9.13).

Листинг 9.13. com.apress.todo.redis.ToDoConsumer.java

```
package com.apress.todo.redis;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

@Component
public class ToDoConsumer {

    private static final Logger log =
        LoggerFactory.getLogger(ToDoConsumer.class);
    private ToDoRepository repository;
```

```
    public ToDoConsumer(ToDoRepository repository){
        this.repository = repository;
    }

    public void handleMessage(ToDo toDo) {
        log.info("Consumer> " + toDo);
        log.info("ToDo created> " + this.repository.save(toDo));
    }
}
```

В листинге 9.13 приведен потребитель, который подписывается на тему для поступающих сообщений `ToDo`. Важно понимать, что для использования прослушивателя обязательно нужен метод с названием `handleMessage` (это ограничение, налагаемое при создании `MessageListenerAdapter`).

Конфигурация приложения ToDo

Теперь создадим конфигурацию для нашего приложения ToDo (листинг 9.14).

Листинг 9.14. com.apress.todo.config.ToDoConfig.java

```
package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import com.apress.todo.redis.ToDoConsumer;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.redis.connection.RedisConnectionFactory;
import org.springframework.data.redis.core.RedisTemplate;
import org.springframework.data.redis.listener.PatternTopic;
import org.springframework.data.redis.listener.RedisMessageListenerContainer;
import org.springframework.data.redis.listener.adapter.MessageListenerAdapter;
import org.springframework.data.redis.serializer.Jackson2JsonRedisSerializer;

@Configuration
public class ToDoConfig {

    @Bean
    public RedisMessageListenerContainer container(RedisConnectionFactory
        connectionFactory,
                            MessageListenerAdapter toDoListenerAdapter,
                            @Value("${todo.redis.topic}") String topic) {
        RedisMessageListenerContainer container = new
            RedisMessageListenerContainer();
        container.setConnectionFactory(connectionFactory);
```

```
        container.addMessageListener(toDoListenerAdapter, new
            PatternTopic(topic));
        return container;
    }

    @Bean
    MessageListenerAdapter toDoListenerAdapter(ToDoConsumer consumer) {
        MessageListenerAdapter messageListenerAdapter =
            new MessageListenerAdapter(consumer);
        messageListenerAdapter.setSerializer(new
            Jackson2JsonRedisSerializer<>(ToDo.class));
        return messageListenerAdapter;
    }

    @Bean
    RedisTemplate<String, ToDo> redisTemplate(RedisConnectionFactory
        connectionFactory){
        RedisTemplate<String,ToDo> redisTemplate = new
            RedisTemplate<String,ToDo>();
        redisTemplate.setConnectionFactory(connectionFactory);
        redisTemplate.setDefaultSerializer(new Jackson2JsonRedisSerializer<>(
            ToDo.class));
        redisTemplate.afterPropertiesSet();
        return redisTemplate;
    }
}
```

В листинге 9.14 приведена необходимая для приложения ToDo конфигурация. В этом классе объявлены следующие компоненты Spring.

- Класс `RedisMessageListenerContainer`. Этот класс отвечает за подключение к теме Redis.
- Класс `MessageListenerAdapter`. Этот адаптер использует класс простого Java-объекта в старом стиле (Plain Old Java Object, POJO) для обработки сообщения. Обязательным требованием является название `handleMessage` для обрабатывающего сообщение метода, который получает сообщение из темы в виде экземпляра `ToDo`, вследствие чего требуется также сериализатор.
- Класс `Jackson2JsonRedisSerializer`. Этот сериализатор производит преобразование из экземпляра `ToDo`/в него.
- Класс `RedisTemplate`. Этот класс реализует паттерн `Template` и аналогичен используемым для прочих технологий обмена сообщениями. Для работы с JSON и преобразованием из экземпляров `ToDo`/в них ему требуется сериализатор.

Эти дополнительные настройки необходимы для работы с форматом JSON и корректного преобразования из экземпляров `ToDo`/в них; но вы можете без них обойтись и воспользоваться конфигурацией по умолчанию, для которой требуется сериализуемый объект (например, типа `String`) для отправки. Вместо него можно применить `StringRedisTemplate`.

Добавьте в файл `application.properties` следующее содержимое:

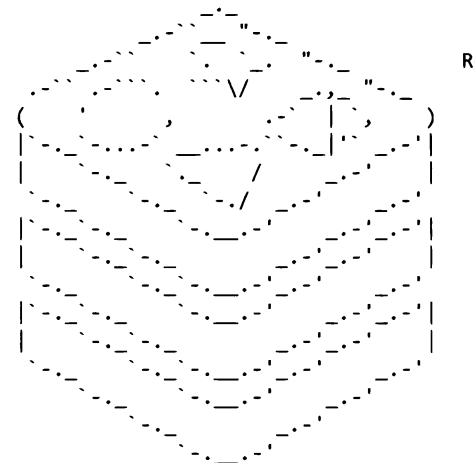
```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop

# ToDo Redis
todo.redis.topic=todos
```

Запуск приложения ToDo

Перед запуском приложения ToDo убедитесь, что сервер Redis запущен. Для запуска сервера выполните в терминале следующую команду:

```
$ redis-server
89887:C 11 Feb 20:17:55.320 # Warning: no config file specified, using the
default config. In order to specify a config file use redis-server /path/
to/redis.conf
89887:M 11 Feb 20:17:55.321 * Increased maximum number of open files to
10032 (it was originally set to 256).
```




```
89887:M 11 Feb 20:17:55.323 # Server started, Redis version 3.0.7
89887:M 11 Feb 20:17:55.323 * The server is now ready to accept connections
on port 6379
```

Этот вывод демонстрирует, что сервер Redis готов к использованию и прослушивает на порте 6379. Откройте новое окно терминала и выполните следующую команду:

```
$ redis-cli
```

Это клиент командной оболочки, который умеет подключаться к серверу Redis. Можете подписаться на тему `todos`, выполнив следующую команду.

```
127.0.0.1:6379> SUBSCRIBE todos
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "todos"
3) (integer) 1
```

Теперь можно запустить приложение как обычно (из IDE или с помощью Maven/Gradle). Если вы используете Maven, выполните:

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

После выполнения этой команды вы должны увидеть в журнале примерно следующее:

```
...
Producer> Message Sent
Consumer> ToDo(id=null, description=workout tomorrow morning!,
 created=null, modified=null, completed=false)
ToDo created> ToDo(id=8a808087645bd67001645bd6785b0000, description=workout
 tomorrow morning!, created=2018-07-02T10:32:19.546, modified=2018-07-
 02T10:32:19.547, completed=false)
...
```

Если теперь заглянуть в командную оболочку Redis, вы должны увидеть там что-то вроде следующего:

```
1) "message"
2) "todos"
3)  "{\"id\":null,\"description\":\"workout tomorrow morning!\",\"created\":
 null,\"modified\":null,\"completed\":false}"
```

И конечно, вы можете увидеть новое запланированное дело в браузере по адресу http://localhost:8080/toDos.

Отлично! Вы создали приложение Spring Boot для обмена сообщениями, использующее Redis. Можете теперь остановить сервер Redis, нажав Ctrl+C.

Удаленный сервер Redis

Для доступа к удаленному серверу Redis необходимо добавить в файл `application.properties` следующие свойства:

```
spring.redis.database=0
spring.redis.host=localhost
spring.redis.password=mysecurepassword
spring.redis.port=6379
```

Вы всегда можете узнать обо всех свойствах Redis в документации Spring Boot по адресу https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html.

Теперь вы знаете, как использовать Redis в качестве брокера сообщений. Если хотите узнать больше о применении хранилищ типа «ключ/значение» со Spring Boot, можете заглянуть на страницу проекта Spring Data Redis: http://projects.spring.io/spring-data-redis/.

Использование WebSockets со Spring Boot

Возможно, логичнее было бы поместить обсуждение WebSockets в главу, посвященную веб-приложениям, но мне кажется, что WebSockets относится скорее к обмену сообщениями, поэтому я поместил данный раздел в эту главу.

WebSockets — новый способ связи, заменяющий веб-технологию «клиент/сервер». Он делает возможным длительные односокетные TCP-соединения между клиентом и сервером. Его также называют технологией *проталкивания* (push), поскольку сервер может отправлять данные без необходимости выполнения клиентом долгих опросов для запроса обновлений.

В этом разделе показан пример, включающий отправку сообщения через конечную точку REST (`Producer`) и получение сообщений (`Consumer`) с помощью веб-страницы и библиотек JavaScript.

Создание приложения ToDo с использованием WebSockets

Создадим приложение ToDo, использующее репозитории JPA REST. Каждое появляющееся новое запланированное дело выводится на веб-странице.

Соединение веб-страницы с приложением ToDo производится посредством WebSockets с помощью протокола STOMP.

Для начала откройте свой любимый браузер и перейдите на сайт Spring Initializr. Добавьте следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-websocket`.
- Name (Название): `todo-websocket`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Websocket`, `Web`, `Lombok`, `JPA`, `REST Repositories`, `H2`, `MySQL`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в вашу любимую IDE (рис. 9.7).

Можете переиспользовать и скопировать/вставить классы `ToDo` и `ToDoRepository`. Необходимо также добавить следующие зависимости; если вы используете Maven, добавьте следующее в файл `pom.xml`:

```
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>sockjs-client</artifactId>
    <version>1.1.2</version>
</dependency>
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>stomp-websocket</artifactId>
    <version>2.3.3</version>
</dependency>

<!-- jQuery -->
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>jquery</artifactId>
    <version>3.1.1</version>
</dependency>

<!-- Bootstrap -->
<dependency>
    <groupId>org.webjars</groupId>
    <artifactId>bootstrap</artifactId>
    <version>3.3.5</version>
</dependency>
```

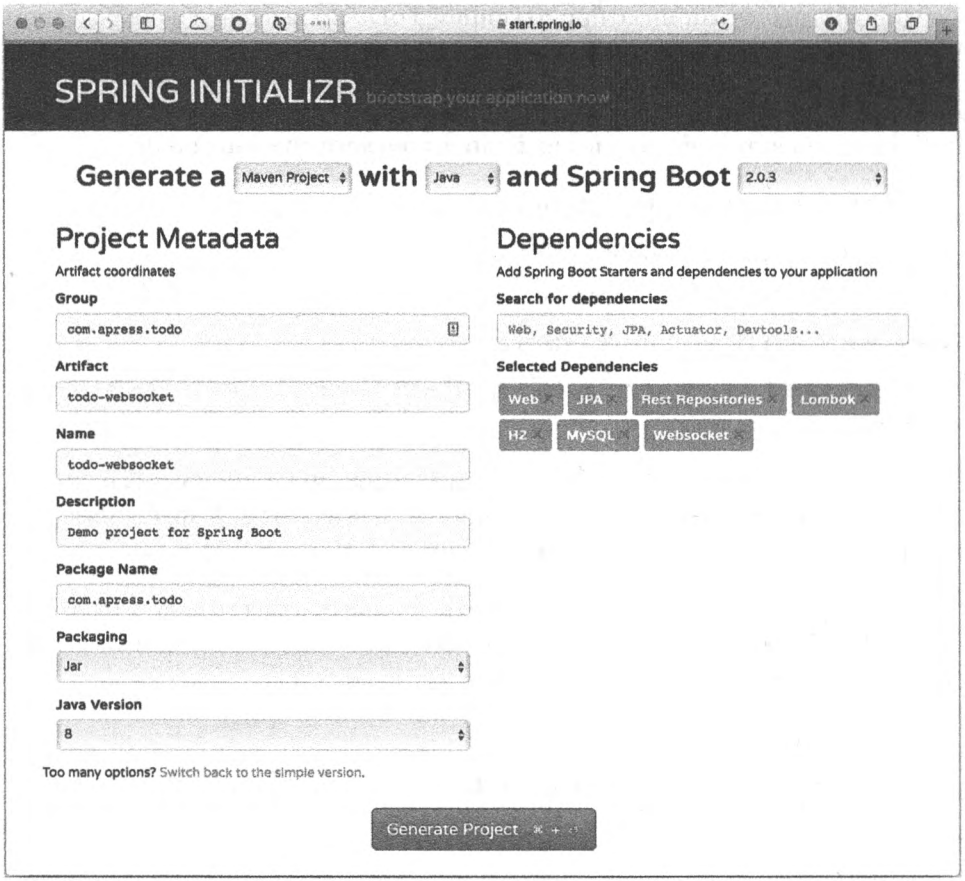


Рис. 9.7. Spring Initializr

Если вы используете Gradle, добавьте следующие зависимости в файл `build.gradle`:

```
compile('org.webjars:sockjs-client:1.1.2')
compile('org.webjars:stomp-websocket:2.3.3')
compile('org.webjars:jquery:3.1.1')
compile('org.webjars:bootstrap:3.3.5')
```

Эти зависимости обеспечивают создание веб-клиента, который нам нужно подключить к брокеру сообщений. Удобнее всего воспользоваться WebJars для включения внешних ресурсов в виде пакетов вместо скачивания их по одному.

Генератор ToDo

Генератор отправляет STOMP-сообщение в тему при отправке нового запланированного дела с помощью HTTP-метода `POST`. Для этого необходимо перехватить событие, выдаваемое Spring Data при сохранении класса предметной области в базе данных.

Во фреймворке Spring Data REST есть несколько событий, позволяющих контролировать происходящее до, во время и после операции сохранения. Создайте класс `ToDoEventHandler` для прослушивания на предмет события `after-create` (листинг 9.15).

Листинг 9.15. com.apress.todo.event.ToDoEventHandler.java

```
package com.apress.todo.event;

import com.apress.todo.config.ToDoProperties;
import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.data.rest.core.annotation.HandleAfterCreate;
import org.springframework.data.rest.core.annotation.RepositoryEventHandler;
import org.springframework.messaging.simp.SimpMessagingTemplate;
import org.springframework.stereotype.Component;

@Component
@RepositoryEventHandler(ToDo.class)
public class ToDoEventHandler {

    private Logger log = LoggerFactory.getLogger(ToDoEventHandler.class);
    private SimpMessagingTemplate simpMessagingTemplate;
    private ToDoProperties toDoProperties;

    public ToDoEventHandler(SimpMessagingTemplate simpMessagingTemplate,
        ToDoProperties toDoProperties){
        this.simpMessagingTemplate = simpMessagingTemplate;
        this.toDoProperties = toDoProperties;
    }

    @HandleAfterCreate
    public void handleToDoSave(ToDo toDo){this.
        simpMessagingTemplate.convertAndSend(this.toDoProperties.getBroker()
         + "/ new",toDo);
        log.info(">> Sending Message to WS: ws://todo/new - " + toDo);
    }
}
```

В листинге 9.15 приведен обработчик событий, получающий событие `afterCreate`. Проанализируем его.

- Аннотация `@RepositoryEventHandler`. Эта аннотация сообщает `BeanPostProcessor` о необходимости просмотреть соответствующий класс на предмет наличия методов-обработчиков.
- Класс `SimpMessagingTemplate`. Еще одна реализация паттерна `Template`, используемая для отправки сообщений с помощью протокола STOMP. Она ведет себя аналогично прочим классам `*Template` из предыдущих разделов.
- Класс `ToDoProperties`. Представляет собой пользовательский обработчик свойств. Описывает брокер (`todo.ws.broker`), конечную точку (`todo.ws.endpoint`) и конечную точку приложения для WebSockets.
- Аннотация `@HandleAfterCreate`. Отмечает метод, получающий все события, происходящие после сохранения класса предметной области в базу данных. Как видите, в нем используется сохраненный в базу данных экземпляр `ToDo`. В этом методе мы задействуем `SimpMessagingTemplate` для отправки экземпляра `ToDo` в конечную точку `/todo/new`. Все, кто подписан на эту конечную точку, получат данное запланированное дело в формате JSON (STOMP).

Далее создадим класс `ToDoProperties` для хранения информации о конечных точках (листинг 9.16).

Листинг 9.16. com.apress.todo.config.ToDoProperties.java

```
package com.apress.todo.config;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

@Data
@ConfigurationProperties(prefix = "todo.ws")
public class ToDoProperties {

    private String app = "/todo-api-ws";
    private String broker = "/todo";
    private String endpoint = "/stomp";
}
```

`ToDoProperties` — вспомогательный класс для хранения информации о брокере (`/stomp`) и о том, куда подключается веб-клиент (тема — `/todo/new`).

Конфигурация приложения ToDo

На этот раз приложение ToDo создает брокер сообщений, принимающий их через WebSockets и использующий протокол STOMP для обмена ими.

Создайте класс конфигурации (листинг 9.17).

Листинг 9.17. com.apress.todo.config.ToDoConfig.java

```
package com.apress.todo.config;

import org.springframework.boot.context.properties.
  EnableConfigurationProperties;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import org.springframework.web.socket.config.annotation.
  EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.
  StompEndpointRegistry;
import org.springframework.web.socket.config.annotation.
  WebSocketMessageBrokerConfigurer;

@Configuration
@EnableWebSocketMessageBroker
@EnableConfigurationProperties(ToDoProperties.class)
public class ToDoConfig implements WebSocketMessageBrokerConfigurer {

    private ToDoProperties props;

    public ToDoConfig(ToDoProperties props){
        this.props = props;
    }

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
         registry.addEndpoint(props.getEndpoint()).setAllowedOrigins("*").
         withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry config) {
        config.enableSimpleBroker(props.getBroker());
        config.setApplicationDestinationPrefixes(props.getApp());
    }
}
```

В листинге 9.17 приведен класс `ToDoConfig`. Проанализируем его.

- Аннотация `@Configuration`. Вы уже знаете, что эта аннотация отмечает класс как конфигурацию для контейнера Spring.

- Аннотация `@EnableWebSocketMessageBroker`. Эта аннотация использует автоконфигурацию для создания всех артефактов, необходимых для обмена сообщениями через WebSockets с помощью брокера, на основе весьма высокоуровневого субпротокола обмена сообщениями. Для настройки конечных точек под свои нужды необходимо переопределить методы интерфейса `WebSocketMessageBrokerConfigurer`.
- Интерфейс `WebSocketMessageBrokerConfigurer`. Его методы переопределяются в нашем классе конфигурации для настройки протоколов и конечных точек под наши нужды.
- Метод `registerStompEndpoints(StompEndpointRegistry registry)`. Регистрирует протокол STOMP; в данном случае он регистрирует конечную точку `/stomp`, используя JavaScript-библиотеку SockJS (https://github.com/sockjs).
- Метод `configureMessageBroker(MessageBrokerRegistry config)`. Выполняет настройку опций брокера сообщений. В данном случае активирует брокер в конечной точке `/todo`. Это значит, что клиентам для использования брокера WebSockets необходимо подключаться к `/todo`.

Теперь добавим информацию в файл `application.properties`.

`src/main/resources/application.properties`

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=create-drop

# Rest-репозитории
spring.data.rest.base-path=/api

# WebSocket
todo.ws.endpoint=/stomp
todo.ws.broker=/todo
todo.ws.app=/todo-api-ws
```

В файле `application.properties` объявляется новая конечная точка базового пути REST (`/api`), поскольку клиент представляет собой HTML-страницу, по умолчанию `index.html`; это значит, что Rest-репозитории находятся в конечной точке `/api/*`, а не в корне приложения.

Веб-клиент ToDo

Именно веб-клиент подключается к брокеру сообщений, где подписывается (посредством протокола STOMP) на получение всех новых запланирован-

ных дел. Этот клиент может быть любого типа, лишь бы он мог работать с WebSockets и знал протокол STOMP.

Создадим простую страницу `index.html`, которая будет подключаться к брокеру (листинг 9.18).

Листинг 9.18. src/main/resources/static/index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>ToDo WebSockets</title>
    <link rel="stylesheet"
          href="/webjars/bootstrap/3.3.5/css/bootstrap.min.css">
    <link rel="stylesheet"
          href="/webjars/bootstrap/3.3.5/css/bootstrap-theme.min.css">
</head>
<body>
<div class="container theme-showcase" role="main">
    <div class="jumbotron">
        <h1>What ToDo?</h1>
        <p>An easy way to find out what your are going to do NEXT!</p>
    </div>
    <div class="page-header">
        <h1>Everybody ToDo's</h1>
    </div>
    <div class="row">
        <div class="col-sm-12">
            <div class="panel panel-primary">
                <div class="panel-heading">
                    <h3 class="panel-title">ToDo:</h3>
                </div>
                <div class="panel-body">
                    <div id="output">
                    </div>
                </div>
            </div>
        </div>
    </div>
</div>

<script src="/webjars/jquery/3.1.1/jquery.min.js"></script>
<script src="/webjars/sockjs-client/1.1.2/sockjs.min.js"></script>
<script src="/webjars/stomp-websocket/2.3.3/stomp.min.js"></script>

<script>

    $(function(){
        var stompClient = null;
```

```
        var socket = new SockJS('http://localhost:8080/stomp');
        stompClient = Stomp.over(socket);

        stompClient.connect({}, function (frame) {
            console.log('Connected: ' + frame);

            stompClient.subscribe('/todo/new', function (data) {
                console.log('>>>>> ' + data);
                var json = JSON.parse(data.body);
                 var result = "<span><strong>[" + json.created +
                     "]</strong> " + json.description + "</span><br/>";
                $("#output").append(result);
            });
        });
    });
</script>
</body>
</html>
```

В листинге 9.18 приведен файл `index.html` — клиент, использующий класс `SockJS` для подключения к конечной точке `/stomp`. Он подписывается на тему `/todo/new` и ожидает добавления в список нового запланированного дела. Применяются также ресурсы WebJars в виде ссылок на библиотеки JavaScript и CSS.

Запуск приложения ToDo

Мы готовы к запуску приложения ToDo. Можете запустить его обычным образом, с помощью IDE или из командной строки. Если вы используете Maven, выполните следующее:

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

Если вы используете Gradle, выполните:

```
$ ./gradlew bootRun
```

Откройте браузер и перейдите по адресу http://localhost:8080. Вы должны увидеть пустое поле для списка ToDo. Откройте терминал и выполните следующие команды:

```
$ curl -XPOST -d '{"description":"Learn to play Guitar"}' -H "Content-Type:
application/json" http://localhost:8080/api/toDos
$ curl -XPOST -d '{"description":"read Spring Boot Messaging book from
Apress"}' -H "Content-Type: application/json" http://localhost:8080/api/toDos
```

Можете добавить еще, если хотите. Теперь в браузере будет виден список запланированных дел (рис. 9.8).

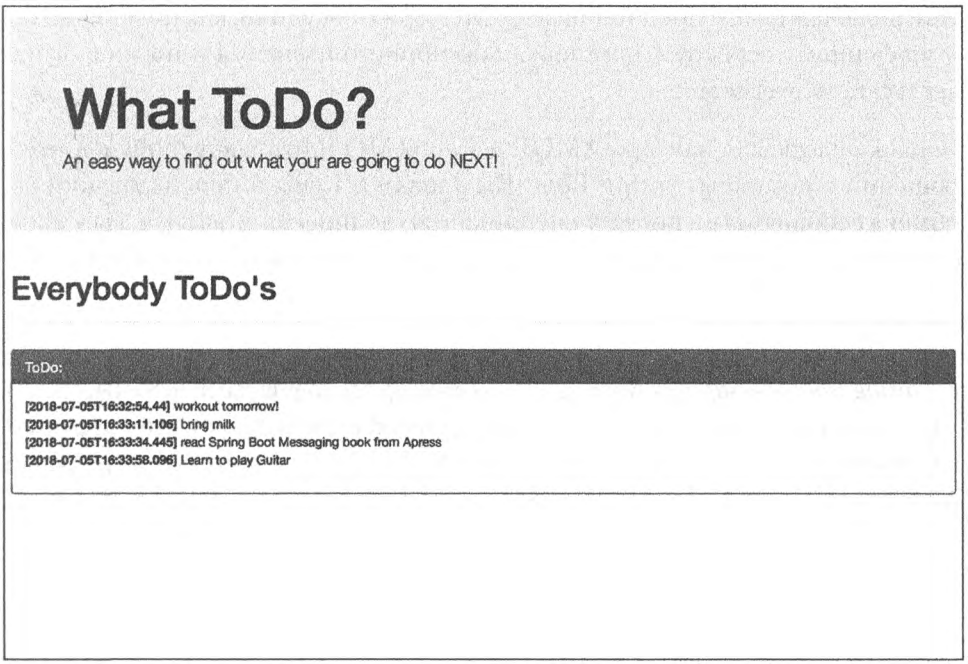


Рис. 9.8. Сообщения SockJS и STOMP: список запланированных дел

На рис. 9.8 показан результат отправки сообщений через WebSockets. Представьте себе возможности, открывающиеся для новых приложений, требующих уведомления в режиме реального времени (например, при создании чатов в реальном времени, обновления курсов акций на лету или обновления сайта без предварительного просмотра или перезагрузки страницы). Все это можно делать с помощью Spring Boot и WebSockets.

ПРИМЕЧАНИЕ

Весь код приложений доступен на веб-сайте Apress. Свежую его версию можно также найти по адресу https://github.com/felipeg48/pro-spring-boot-2nd.

Резюме

В этой главе мы обсудили все используемые для обмена сообщениями технологии, включая JavaScript и RabbitMQ. Мы также обсудили, как подключиться к удаленному серверу с помощью указания его названия и порта в файле `application.properties`.

Я также рассказал вам про AMQP и RabbitMQ и отправку/получение сообщений с помощью Spring Boot. Вы узнали о Redis и использовании его обмена сообщениями по типу «публикация/подписка». Наконец, вы узнали о протоколе WebSockets и о том, насколько просто использовать его с помощью Spring Boot.

Если вас интересует вопрос обмена сообщениями, можете прочитать мою книгу *Spring Boot Messaging* (Apress, 2017) (www.apress.com/us/book/9781484212257), в которой эта тема обсуждается подробнее и показано больше паттернов обмена сообщениями, от простых событий приложения до облачных программных решений на основе Spring Cloud Stream и его транспортных абстракций.

В следующей главе мы обсудим Spring Boot Actuator и мониторинг приложений Spring Boot.

10 Spring Boot Actuator

В этой главе обсуждается модуль Spring Boot Actuator и объясняется, как использовать его возможности для мониторинга приложений Spring Boot.

Все разработчики нередко начинают во время и после разработки проверять журналы. Проверяют, работает ли бизнес-логика так, как должна, время обработки сервисов и т. д. Даже полный набор всех модульных, интеграционных и регрессионных тестов не гарантирует полного отсутствия внешних сбоев, включая сетевые (сбои соединений, проблемы со скоростью и т. д.), дисковые (проблемы со свободным пространством, правами доступа и т. д.) и пр.

При развертывании приложения в промышленной эксплуатации эти вопросы стоят еще более остро. Необходимо внимательно следить за приложением, а иногда и системой в целом. В случае зависимости от нефункциональных требований, например, систем мониторинга состояния различных приложений, возможно, с выдачей предупреждений в случае превышения приложением определенного порогового значения или, хуже того, его сбоя необходимо действовать как можно быстрее.

Разработчики в своей работе зависят от множества сторонних технологий, и я не утверждаю, что это плохо, просто это значит, что вся тяжесть работы ложится на плечи специалистов по интеграции разработки и эксплуатации. Им приходится следить за каждым из приложений в отдельности и за системой в целом.

Возможности модуля

Spring Boot включает модуль Actuator, обеспечивающий проверку соблюдения нефункциональных требований к приложению. Модуль Spring Boot Actuator предоставляет готовые возможности мониторинга метрик и аудита.

Особенно привлекательным для разработчика делает модуль Actuator возможность обеспечивать доступность данных посредством различных технологий, таких как HTTP (конечные точки) и JMX. Мониторинг метрик модуля Spring Boot Actuator можно выполнять через фреймворк Micrometer (http://micrometer.io/), благодаря чему можно один раз написать код метрик и использовать его в любом не зависящем от поставщика движке, например Prometheus, Atlas, CloudWatch, Datadog и многих, многих других.

Создание приложения ToDo с использованием Actuator

Начнем с использования модуля Spring Boot Actuator в приложении ToDo, чтобы лучше понять, как он работает. Можете создавать приложение с нуля или просто следить за кодом в следующих разделах. Если начинаете с нуля, то перейдите в Spring Initializr (https://start.spring.io) и внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-actuator`.
- Name (Название): `todo-actuator`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Web`, `Lombok`, `JPA`, `REST Repositories`, `Actuator`, `H2`, `MySQL`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами интегрированную среду разработки (рис. 10.1).

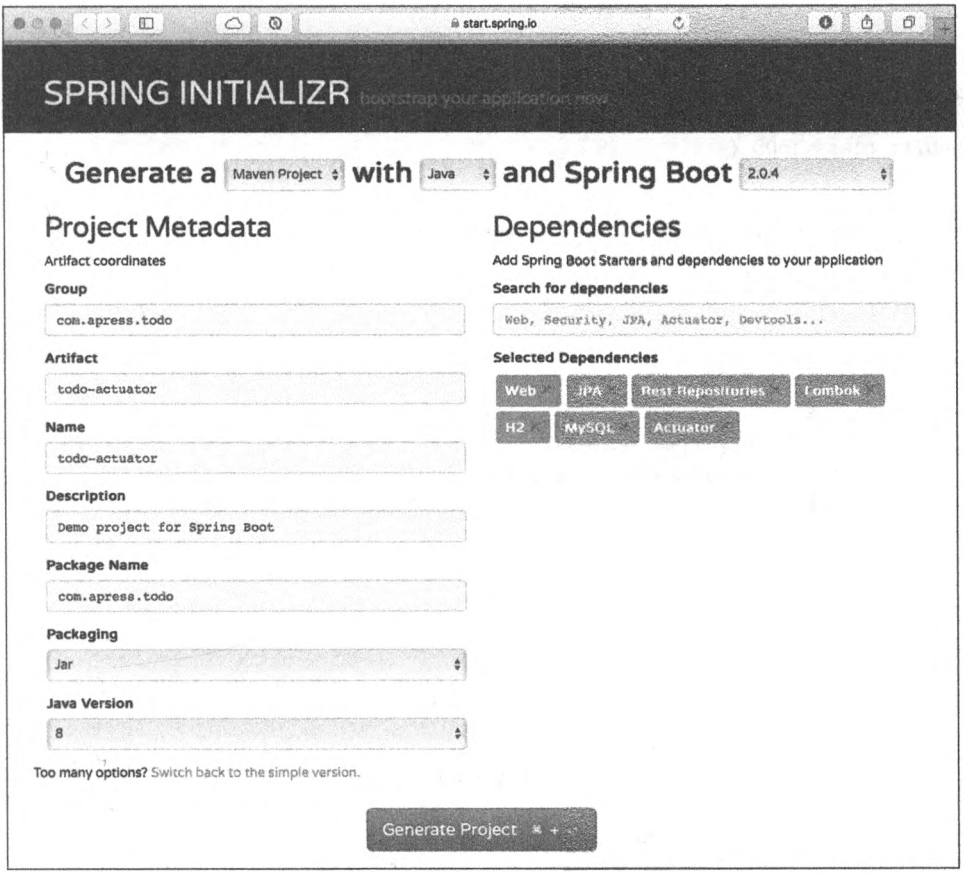


Рис. 10.1. Spring Initializr

Пока здесь нет ничего отличного от других проектов; единственная новая зависимость — модуль Actuator. Можете скопировать/переиспользовать класс предметной области `ToDo` и интерфейс `ToDoRepository` (листинги 10.1 и 10.2).

Листинг 10.1. com.apress.todo.domain.ToDo.java

```
package com.apress.todo.domain;

import lombok.Data;
import org.hibernate.annotations.GenericGenerator;

import javax.persistence.*;
import javax.validation.constraints.NotBlank;
```

```
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;

@Entity
@Data
public class ToDo {

    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    @NotNull
    @NotBlank
    private String description;

    @Column(insertable = true, updatable = false)
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;

    public ToDo(){}
    public ToDo(String description){
        this.description = description;
    }

    @PrePersist
    void onCreate() {
        this.setCreated(LocalDateTime.now());
        this.setModified(LocalDateTime.now());
    }

    @PreUpdate
    void onUpdate() {
        this.setModified(LocalDateTime.now());
    }
}
```

Листинг 10.2. com.apress.todo.repository.ToDoRepository.java

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends CrudRepository<ToDo,String> { }
```

Прежде чем запустить приложение ToDo, обратите внимание, есть ли в вашем файле `pom.xml` (если вы используете Maven) или в файле `build.gradle` (если вы используете Gradle) зависимость `spring-boot-starter-actuator`.

Можете теперь запустить приложение ToDo. Важно обратить внимание на выводимое в журнал, которое должно выглядеть примерно следующим образом.

```
INFO 41925 --- [main] s... : Mapped "{[/actuator/health],methods=[GET],
 produces=[application/vnd.spring-boot.actuator.v2+json || application/
 json]}" ...
INFO 41925 --- [main] s... : Mapped "{[/actuator/info],methods=[GET],
 produces=[application/vnd.spring-boot.actuator.v2+json || application/
 json]}" ...
INFO 41925 --- [main] s... : Mapped "{[/actuator],methods=[GET],produces=
 [application/vnd.spring-boot.actuator.v2+json || application/json]}" ...
```

По умолчанию модуль Actuator открывает три доступные для посещения конечные точки.

- `/actuator/health`. Эта конечная точка предоставляет доступ к основной информации о состоянии приложения. При обращении через браузер или из командной строки вы получите следующий ответ:

  ```
  {
      "status": "UP"
  }
  ```

- `/actuator/info`. Эта конечная точка может отображать произвольную информацию приложения. При обращении к ней вы получите пустой ответ, но если добавить следующее в файл `application.properties`:

  ```
  spring.application.name=todo-actuator
  info.application-name=${spring.application.name}
  info.developer.name=Awesome Developer
  info.developer.email=awesome@example.com
  ```

  то вы получите такой ответ:

  ```
  {
      "application-name": "todo-actuator",
      "developer": {
          "name": "Awesome Developer",
          "email": "awesome@example.com"
      }
  }
  ```

- `/actuator`. Префикс для всех конечных точек актуатора. Если зайти на нее с помощью браузера или из командной строки, вы получите следующий ответ:

  ```
  {
    "_links": {
      "self": {
  ```

```
            "href": "http://localhost:8080/actuator",
            "templated": false
        },
        "health": {
            "href": "http://localhost:8080/actuator/health",
            "templated": false
        },
        "info": {
            "href": "http://localhost:8080/actuator/info",
            "templated": false
        }
    }
}
```

По умолчанию все конечные точки (существуют не только эти, но и другие) включены, за исключением конечной точки `/actuator/shutdown`, но почему же доступны только две (состояние приложения и информация)? Фактически все они доступны через JMX, но, поскольку некоторые из них содержат требующую защиты информацию, важно четко понимать, какую информацию можно делать видимой через веб.

Существуют два свойства, с помощью которых можно открыть к ним веб-доступ: `management.endpoints.web.exposure.include` и `management.endpoints.web.exposure.exclude`. Конечные точки можно перечислять по отдельности, разделенные запятыми, либо включить все с помощью `*`.

То же самое относится и к открытию доступа к конечным точкам через JMX с помощью свойств `management.endpoints.jmx.exposure.include` и `management.endpoints.jmx.exposure.exclude`. Помните, что по умолчанию все конечные точки доступны через JMX.

Как я уже упоминал ранее, можно не только открывать доступ к конечным точкам, но и делать их активными. Для этого используется следующий синтаксис: `management.endpoint.<НАЗВАНИЕ_КОНЕЧНОЙ_ТОЧКИ>.enabled`. Так, чтобы сделать активной конечную точку `/actuator/shutdown` (по умолчанию она отключена), необходимо вставить в файл `application.properties` следующее:

```
management.endpoint.shutdown.enabled=true
```

Можете вставить такое свойство в файл `application.properties`, чтобы сделать доступными все конечные точки веб-актуатора:

```
management.endpoints.web.exposure.include=*
```

Если теперь взглянуть на вывод конечной точки `/actuator`, вы увидите дополнительные конечные точки, например `/actuator/beans`, `/actuator/conditions` и т. д. Обсудим некоторые из них подробнее.

/actuator

Конечная точка `/actuator` представляет собой префикс для всех конечных точек, но если обратиться к ней, можно получить гипермедиастраницу со списком всех остальных конечных точек. Так, если вы перейдете по адресу http://localhost:8080/actuator, то увидите нечто напоминающее рис. 10.2.

Рис. 10.2. http://localhost:8080/actuator

/actuator/conditions

Эта конечная точка отображает отчет об автоконфигурации. В нем можно видеть две группы: `positiveMatches` и `negativeMatches`[1]. Как вы помните, основная возможность Spring Boot — автоконфигурация приложения на основе просмотра путей к классам и зависимостей, тесно связанная со стартовыми POM-файлами и добавленными нами в файл `pom.xml` дополнительными зависимостями. Если вы перейдете по адресу http://localhost:8080/actuator/conditions, то увидите нечто напоминающее рис. 10.3.

Рис. 10.3. http://localhost:8080/actuator/conditions

[1] На самом деле три: вышеупомянутые и unconditionalClasses.

/actuator/beans

Эта конечная точка выводит список всех используемых в приложении компонентов Spring. Напомню, что, хотя для создания простого веб-приложения в Spring достаточно нескольких строк кода, Spring создает все необходимые для работы приложения компоненты «за кулисами». Если вы перейдете по адресу http://localhost:8080/actuator/beans, то увидите нечто напоминающее рис. 10.4.

Рис. 10.4. http://localhost:8080/actuator/beans

/actuator/configprops

Эта конечная точка выводит список всех свойств конфигурации, задаваемых компонентами `@ConfigurationProperties` (я демонстрировал это в предыдущих главах). Напомню, что вы можете создавать свои собственные свойства конфигурации, описывая их в файлах `application.properties` или YAML. На рис. 10.5 приведен пример вывода этой конечной точки.

Рис. 10.5. http://localhost:8080/actuator/configprops

/actuator/threaddump

Эта конечная точка запускает дамп потоков выполнения приложения. Она выводит все работающие потоки и их трассу вызовов в выполняющей

приложение JVM. Перейдите по адресу http://localhost:8080/actuator/threaddump (рис. 10.6).

Рис. 10.6. http://localhost:8080/actuator/threaddump

/actuator/env

Эта конечная точка раскрывает все свойства из интерфейса `ConfigurableEnvironment` Spring. В результате демонстрируются все активные профили и системные переменные среды, а также все свойства приложения, включая свойства Spring Boot. Перейдите по адресу http://localhost:8080/actuator/env (рис. 10.7).

Рис. 10.7. http://localhost:8080/actuator/env

/actuator/health

Эта конечная точка отображает информацию о состоянии приложения. По умолчанию выводится общее состояние системы:

```
{
   "status": "UP"
}
```

Чтобы получить дополнительную информацию о других подсистемах, необходимо указать следующее свойство в файле `application.properties`:

```
management.endpoint.health.show-details=always
```

Отредактируйте файл `application.properties` и перезапустите приложение ToDo. При наличии базы данных (как у нас) вы увидите ее состояние, а также по умолчанию `diskSpace` операционной системы. После запуска приложения перейдите по адресу http://localhost:8080/actuator/health (рис. 10.8).

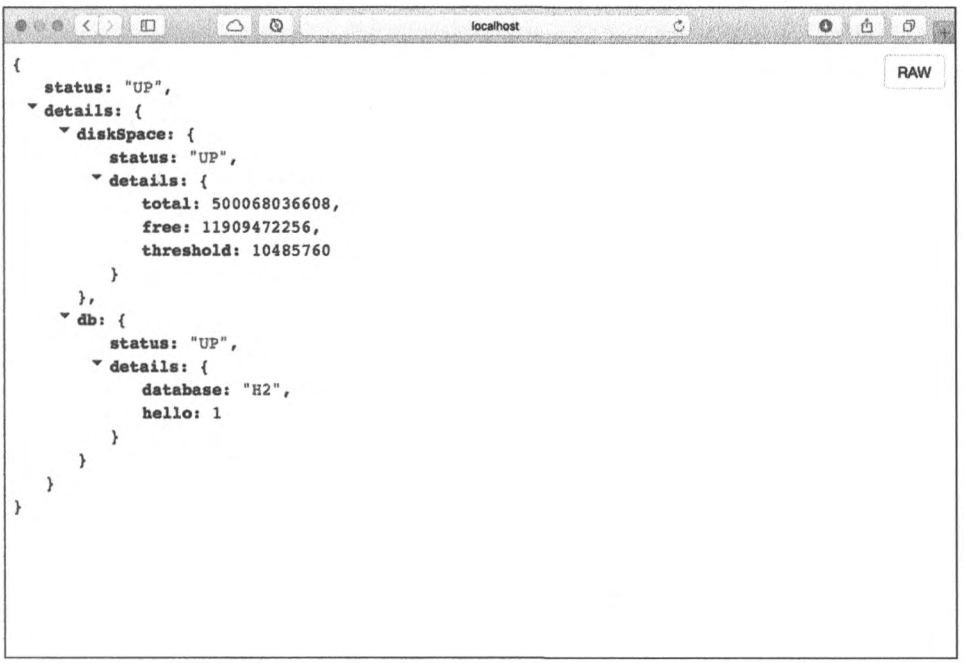


Рис. 10.8. http://localhost:8080/actuator/health

/actuator/info

Эта конечная точка отображает всю общедоступную информацию приложения. Это значит, что данную информацию необходимо добавить в файл `application.properties`, что рекомендуется делать в случае нескольких приложений Spring Boot.

/actuator/loggers

Эта конечная точка отображает список всех средств журналирования в системе. На рис. 10.9 показаны уровни журналирования для конкретных пакетов.

Рис. 10.9. http://localhost:8080/actuator/loggers

/actuator/loggers/{name}

С помощью этой конечной точки можно посмотреть на конкретный пакет и его уровень журналирования. Так, если вы задали, например, настройку `logging.level.com.apress.todo=DEBUG` и перейдете в конечную точку http://localhost:8080/actuator/loggers/com.apress.todo, то получите следующее:

```
{
  "configuredLevel": DEBUG,
  "effectiveLevel": "DEBUG"
}
```

/actuator/metrics

Эта конечная точка демонстрирует информацию о метриках текущего приложения. Здесь вы можете посмотреть, сколько оно задействует оперативной памяти, сколько памяти свободно, время непрерывной работы приложения, размер используемой кучи, число применяемых потоков выполнения и т. д. (рис. 10.10 и 10.11).

Рис. 10.10. http://localhost:8080/actuator/metrics

Для обращения к конкретной метрике необходимо добавить ее название в конец адреса конечной точки; так, чтобы получить больше информации об `jvm.memory.max`, необходимо перейти по адресу http://localhost:8080/actuator/metrics/jvm.memory.max (см. рис. 10.11).

Рис. 10.11. http://localhost:8080/actuator/metrics/jvm.memory.max

Если взглянуть на рис. 10.11, в раздел `availableTags` (имеющиеся теги), можно получить дополнительную информацию, добавив в конец адреса конечной точки `tag=КЛЮЧ:ЗНАЧЕНИЕ`. Например, можно воспользоваться адресом http://localhost:8080/actuator/metrics/jvm.memory.max?tag=area:heap и получить дополнительную информацию о куче.

/actuator/mappings

Эта конечная точка выводит все списки всех объявленных в приложении путей `@RequestMapping`. Это очень удобно, если нужно узнать, какие объявлены

отображения. Если ваше приложение запущено, перейдите к конечной точке http://localhost:8080/actuator/mappings (рис. 10.12).

Рис. 10.12. http://localhost:8080/actuator/mappings

/actuator/shutdown

По умолчанию эта конечная точка неактивна. Она позволяет аккуратно завершить работу приложения. Данную конечную точку можно (и нужно) использовать защищенным образом. Для включения конечной точки `/actuator/shutdown` необходимо добавить в `application.properties` следующее:

```
management.endpoint.shutdown.enabled=true
```

Разумно будет обеспечить безопасность этой конечной точки. Добавьте зависимость `spring-boot-starter-security` в файл `pom.xml` (если вы используете Maven):

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Если вы используете Gradle, можете добавить следующую зависимость в файл `build.gradle`:

```
compile('org.springframework.boot: spring-boot-starter-security')
```

Не забывайте, что добавление указанной зависимости приводит к тому, что безопасность по умолчанию включена. Имя пользователя `user` и пароль выводятся в журнал. Можно также повысить степень безопасности с помощью механизмов аутентификации в оперативной памяти, на основе базы данных или LDAP (см. дополнительную информацию в главе 8).

Пока что добавьте `management.endpoint.shutdown.enabled=true` и зависимость `spring-boot-starter-security` и перезапустите приложение. После запуска приложения посмотрите в журнал и сохраните выведенный туда пароль для использования в конечной точке `/actuator/shutdown`.

```
...
Using default security password: 2875411a-e609-4890-9aa0-22f90b4e0a11
...
```

Теперь можно открыть терминал и выполнить следующую команду:

```
$ curl -i -X POST http://localhost:8080/shutdown -u user:2875411a-e609-
 4890-9aa0-22f90b4e0a11
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
X-Frame-Options: DENY
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
X-Application-Context: application
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 17 Feb 2018 04:22:58 GMT

{"message":"Shutting down, bye..."}
```

Как видно из этого вывода, для обращения к конечной точке `/actuator/shutdown` используется метод `POST` и передаются выведенные в журнал имя

пользователя и пароль. В результате выводится сообщение `Shutting down, bye...`. И конечно, работа приложения завершается. Опять же важно отдавать себе отчет в необходимости обеспечения безопасности этой конечной точки.

/actuator/httptrace

Эта конечная точка выводит информацию о трассе вызовов, то есть обычно о нескольких последних HTTP-запросах. С ее помощью можно просматривать всю информацию как запроса, так и возвращаемую в ответ, для отладки приложения на уровне HTTP. Можете запустить приложение и перейти по адресу http://localhost:8080/actuator/httptrace. Вы должны увидеть нечто напоминающее рис. 10.13.

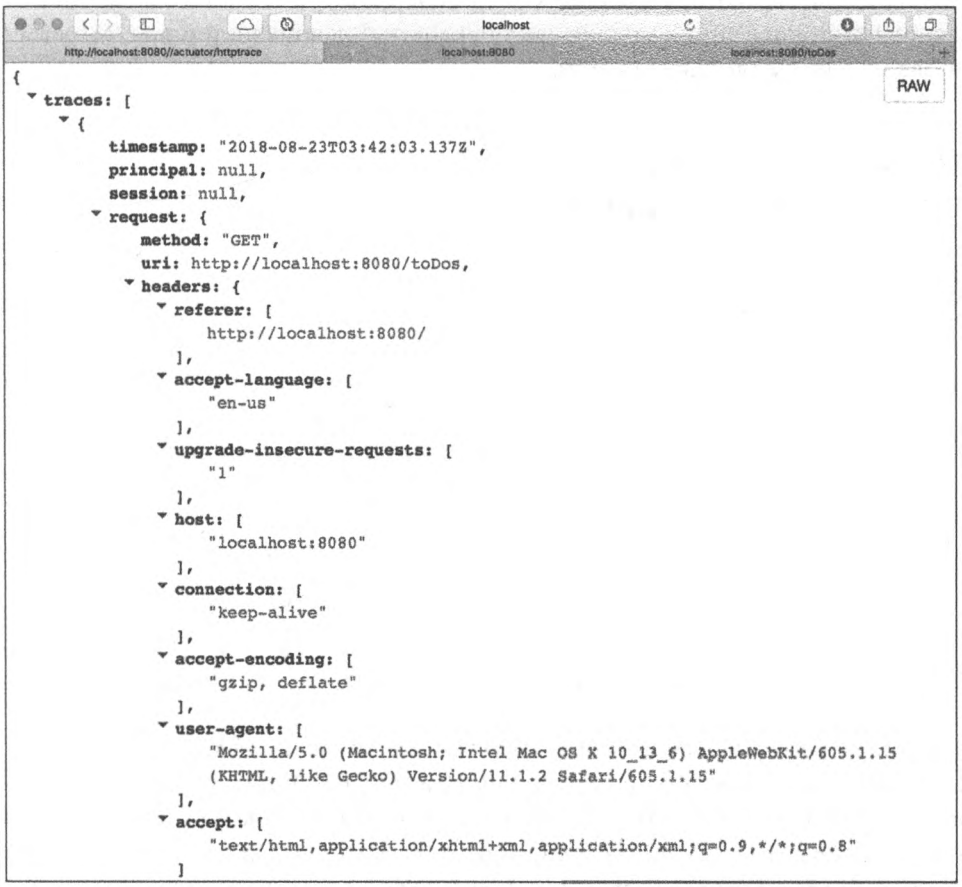


Рис. 10.13. http://localhost:8080/actuator/httptrace

Изменение идентификатора конечной точки

Можно настроить идентификатор конечной точки, изменив таким образом ее название. Допустим, вам не нравится название `/actuator/beans`, указывающее на компоненты Spring, и вы хотели бы поменять название на `/actuator/spring`.

Это изменение можно произвести в файле `application.properties`, указав там `management.endpoints.web.path-mapping.<название_конечной_точки>=<новое_название>`, например, вот так: `management.endpoints.web.path-mapping.beans=spring`.

Если перезапустить приложение (остановить и запустить снова, чтобы изменения вступили в силу), можно будет обращаться к конечной точке `/actuator/beans` по адресу `/actuator/spring`.

Поддержка CORS в Spring Boot Actuator

С помощью модуля Spring Boot Actuator можно настроить совместное использование ресурсов между разными источниками (Cross-Origin Resource Sharing, CORS), позволяющее указывать, каким доменам разрешено задействовать конечную точку актуатора. Обычно это делают для доступа других приложений к конечным точкам, которые из соображений безопасности должны иметь возможность выполнять только авторизованные домены.

Задать эти настройки можно в файле `application.properties`.

```
management.endpoints.web.cors.allowed-origins=http://mydomain.com
management.endpoints.web.cors.allowed-methods=GET, POST
```

Если ваше приложение работает, остановите его и запустите снова.

Обычно в свойстве `management.endpoints.web.cors.allowed-origins` следует указывать название домена вроде http://mydomain.com или, например, http://localhost:9090, а не `*`, благодаря чему возможен доступ к конечным точкам без риска взлома сайта. Это очень похоже на использование аннотации `@CrossOrigin(origins = "http://localhost:9000")` в контроллере.

Изменение пути конечных точек управления приложением

По умолчанию управление Spring Boot Actuator осуществляется через конечную точку `/actuator` как корневую в том смысле, что ко всем конечным точкам актуатора можно получить доступ через `/actuator`, например, `/actuator/beans`, `/actuator/health` и т. д. Прежде чем продолжать, остановите приложение. Поменять путь контекста управления можно с помощью добавления в файл `application.properties` следующего свойства:

```
management.endpoints.web.base-path=/monitor
```

Если теперь снова запустить приложение, вы увидите, что `EndpointHandlerMapping` теперь отображает все конечные точки на пути контекста вида `/monitor/<название_конечной_точки>`. Теперь к конечной точке `/httptrace` можно обратиться через http://localhost:8080/monitor/httptrace.

Можно также поменять адрес сервера, добавить SSL, использовать какой-либо конкретный IP-адрес или поменять порт для конечных точек с помощью свойств `management.server.*`.

```
management.server.servlet.context-path=/admin
management.server.port=8081
management.server.address=127.0.0.1
```

У этой конфигурации есть своя конечная точка с `context-path` вида `/admin/actuator/<название_конечной_точки>`. При этом используется порт 8081 (это значит, что прослушивается два порта: 8080 для приложения и 8081 для конечных точек управления). Конечные точки управления привязаны к адресу 127.0.0.1.

Отключить конечные точки (из соображений безопасности) можно двумя способами: либо с помощью свойства `management.endpoints.enabled-by-default=false`, либо воспользовавшись `management.server.port=-1`.

Обеспечение безопасности конечных точек

Обеспечить безопасность конечных точек актуатора можно, включив зависимость `spring-boot-starter-security` и настроив `WebSecurityConfigurerAdapter`, а также с помощью конфигураций `HttpSecurity` и `RequestMatcher`.

```
@Configuration
public class ToDoActuatorSecurity extends WebSecurityConfigurerAdapter {
```

```
    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
        .requestMatcher(EndpointRequest.toAnyEndpoint())
        .authorizeRequests()
        .anyRequest().hasRole("ENDPOINT_ADMIN")
        .and()
        .httpBasic();
    }
}
```

Теперь для доступа к конечным точкам необходима роль `ENDPOINT_ADMIN`.

Настройка конечных точек

По умолчанию кэш конечных точек актуатора отвечает на операции чтения без параметров; для изменения такого поведения можно использовать свойство `management.endpoint.<endpoint-name>.cache.time-to-live`. Например, если нужно изменить кэш `/actuator/beans`, можно добавить следующее в файл `application.properties`:

```
management.endpoint.beans.cache.time-to-live=10s
```

Реализация пользовательских конечных точек актуатора

Конечную точку актуатора можно расширить или создать свою, пользовательскую. Для этого необходимо снабдить соответствующий класс аннотацией `@Endpoint`, а его методы — аннотациями `@ReadOperation`, `@WriteOperation` или `@DeleteOperation`; по умолчанию ваша конечная точка будет доступна через JMX и по HTTP.

Если конечная точка должна быть доступна только через JMX, можно снабдить класс аннотацией `@JmxEndpoint`. Если же нужен только веб-доступ, можно использовать для класса аннотацию `@WebEndpoint`.

При создании методов можно принимать параметры, преобразуемые в правильный тип, необходимый экземпляру `ApplicationConversionService`, которые используют типы контента `application/vnd.spring-boot.actuator.v2+json` и `application/json`.

Возвращать в сигнатуре любого метода можно любой тип (даже `void` или `Void`). Обычно возвращаемый тип контента зависит от возвращаемого типа метода. Если метод возвращает `org.springframework.core.io.Resource`, то тип контента будет `application/octet-stream`; для всех остальных типов возвращаемые типы контента — `application/vnd.spring-boot.actuator.v2+json`, `application/json`.

При применении посредством веб-доступа пользовательских конечных точек актуатора каждой операции соответствует свой HTTP-метод: `@ReadOperation(Http.GET)`, `@WriteOperation(Http.POST)` и `@DeleteOperation(Http.DELETE)`.

Создание приложения ToDo с пользовательскими конечными точками актуатора

Создадим пользовательскую конечную точку (`/todo-stats`) для отображения общего количества запланированных дел в базе данных и количества завершенных. Создадим также операцию записи для завершения запланированного дела и даже операцию для его удаления из базы данных.

Теперь создадим класс `ToDoStatsEndpoint`, который будет включать всю логику нашей пользовательской конечной точки (листинг 10.3).

Листинг 10.3. com.apress.todo.actuator.ToDoStatsEndpoint.java

```
package com.apress.todo.actuator;

import com.apress.todo.domain.ToDo;
import com.apress.todo.repository.ToDoRepository;
import lombok.AllArgsConstructor;
import lombok.Data;
import org.springframework.boot.actuate.endpoint.annotation.*;
import org.springframework.stereotype.Component;

@Component
@Endpoint(id="todo-stats")
public class ToDoStatsEndpoint {

    private ToDoRepository toDoRepository;

    ToDoStatsEndpoint(ToDoRepository toDoRepository){
        this.toDoRepository = toDoRepository;
    }
```

```
    @ReadOperation
    public Stats stats() {
         return new Stats(this.toDoRepository.count(),
                          this.toDoRepository.countByCompleted(true));
    }

    @ReadOperation
    public ToDo getToDo(@Selector String id) {
        return this.toDoRepository.findById(id).orElse(null);
    }

    @WriteOperation
    public Operation completeToDo(@Selector String id) {
        ToDo toDo = this.toDoRepository.findById(id).orElse(null);
        if(null != toDo){
            toDo.setCompleted(true);
            this.toDoRepository.save(toDo);
            return new Operation("COMPLETED",true);
        }
        return new Operation("COMPLETED",false);
    }

    @DeleteOperation
    public Operation removeToDo(@Selector String id) {
        try {
            this.toDoRepository.deleteById(id);
            return new Operation("DELETED",true);
        }catch(Exception ex){
            return new Operation("DELETED",false);
        }
    }

    @AllArgsConstructor
    @Data
    public class Stats {
        private long count;
        private long completed;
    }

    @AllArgsConstructor
    @Data
    public class Operation{
        private String name;
        private boolean successful;
    }
}
```

В листинге 10.3 показана пользовательская конечная точка, выполняющая такие операции, как отображение статистики (общего числа запланированных дел и количества завершенных дел), получение объекта ToDo, удаление его и перевод его в состояние завершенного. Проанализируем ее.

- `@Endpoint`. Указывает, что данный тип является конечной точкой актуатора, выдающей информацию о запущенном приложении. Доступ к конечным точкам возможен посредством множества технологий, включая JMX и HTTP. В данном случае конечной точкой для актуатора является класс `ToDoStatsEndpoint`.
- `@ReadOperation`. Указывает, что метод конечной точки представляет собой операцию чтения (обеспечивает возврат классом запланированного дела по его идентификатору).
- `@Selector`. Эта аннотация для параметра метода конечной точки указывает, что этот параметр выбирает подмножество данных конечной точки. В нашем случае играет роль способа модификации значения для обновления состояния запланированного дела с целью отметить его как завершенное.
- `@WriteOperation`. Указывает, что метод конечной точки представляет собой операцию записи. Очень похоже на событие `POST`.
- `@DeleteOperation`. Указывает, что метод конечной точки представляет собой операцию удаления.

Листинг 10.3 очень напоминает REST API, но в данном случае все методы доступны посредством протокола JMX. Кроме того, метод `stats` использует объект `toDoRepository` для вызова метода `countByCompleted`. Добавим его в интерфейс `ToDoRepository` (листинг 10.4).

Листинг 10.4. com.apress.todo.repository.ToDoRepository.java — версия 2

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends CrudRepository<ToDo,String> {
    public long countByCompleted(boolean completed);
}
```

В листинге 10.4 приведена версия 2 интерфейса `ToDoRepository`. Теперь в нем есть объявление нового метода, `countByCompleted`; помните, что это метод для *именованного запроса* и создание соответствующего оператора SQL для подсчета числа завершенных запланированных дел производит модуль Spring Data.

Как вы видите, создание пользовательской конечной точки не представляет трудностей. Если теперь запустить приложение и перейти по адресу http://localhost:8080/actuator, вы увидите в списке конечную точку `todo-stats` (рис. 10.14).

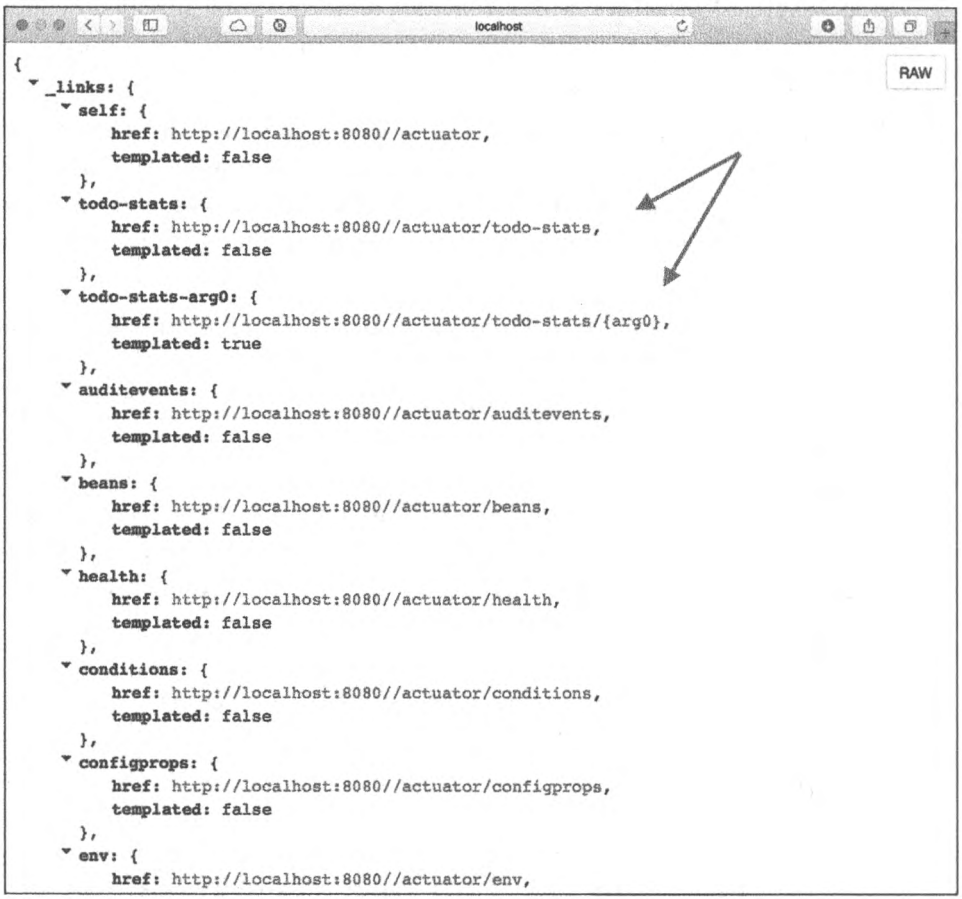


Рис. 10.14. http://localhost:8080/actuator — пользовательская конечная точка todo-stats

Если щелкнуть на ссылке для `todo-stats`, вы увидите показанное на рис. 10.15.

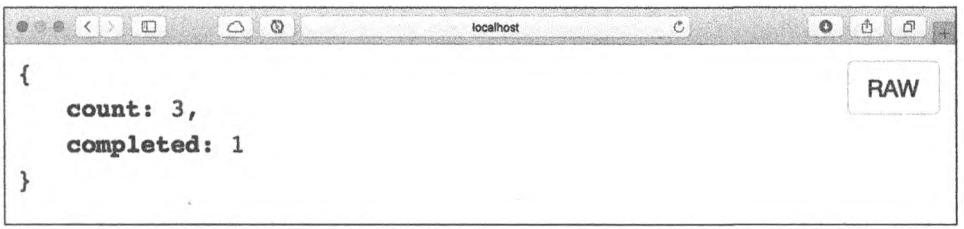
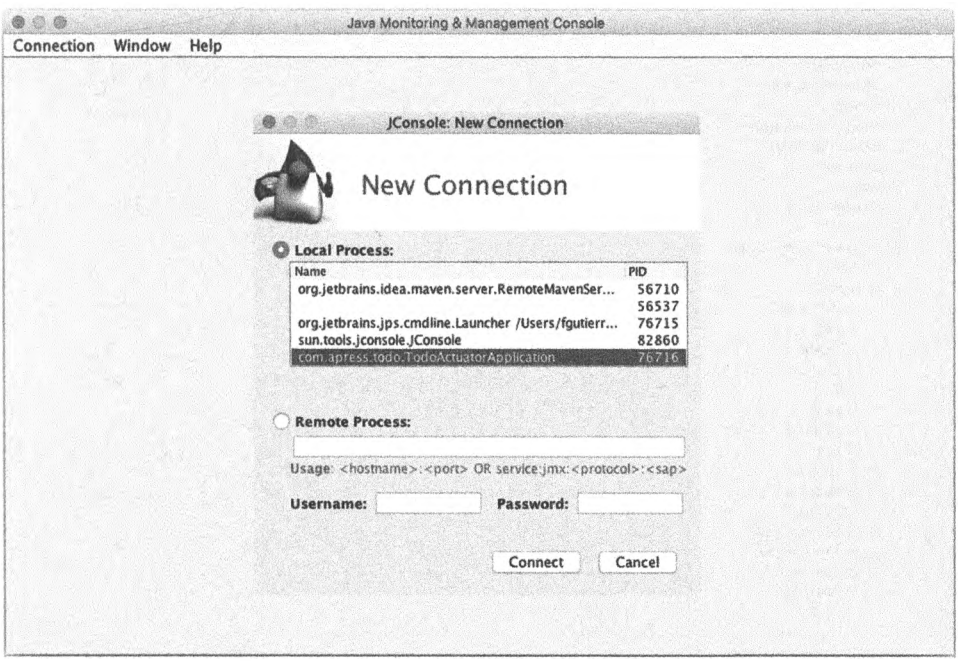


Рис. 10.15. http://localhost:8080/actuator/todo-stats

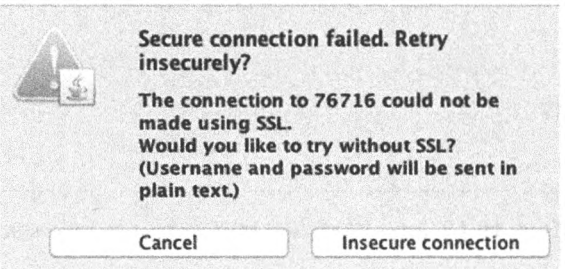
Совсем просто, правда? Но как насчет других операций? Попробуем их. Для этого воспользуемся JMX и JConsole (устанавливается вместе с JDK[1]). Откройте окно терминала и выполните команду `jconsole`.

1. Выберите из списка `com.apress.todo.ToDoActuatorApplication` и нажмите Connect (Подключиться).

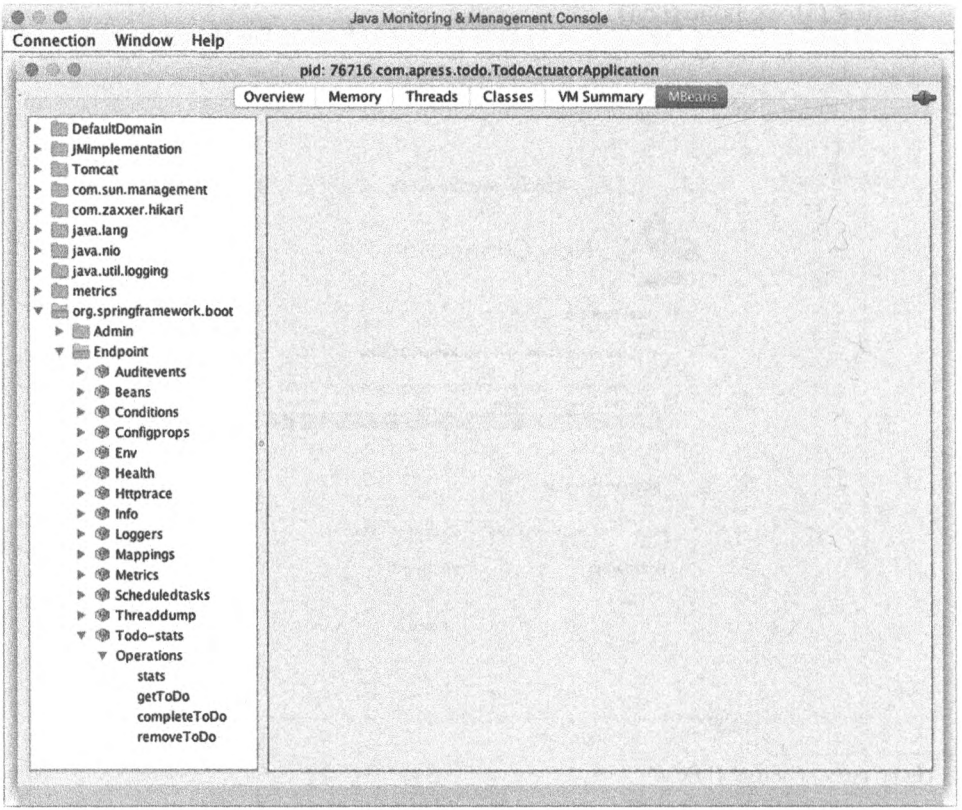


[1] Не во всех версиях JDK.

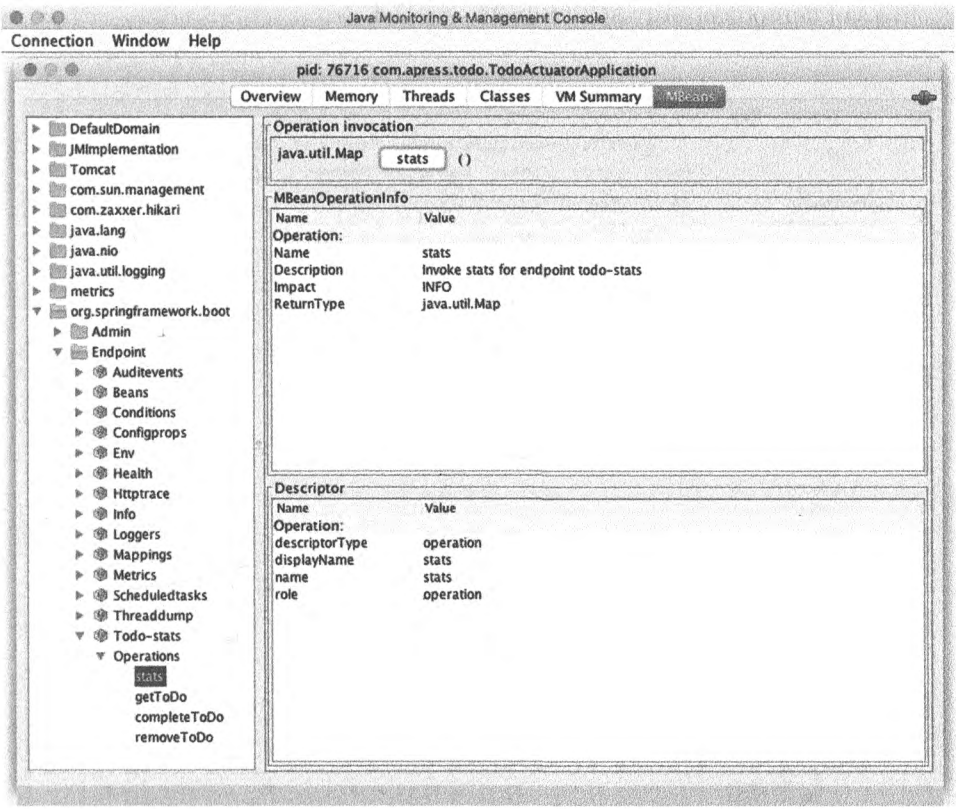
2. Пока возможности защищенного соединения нет, но не бойтесь и нажмите кнопку Insecure Connection (Небезопасное соединение).



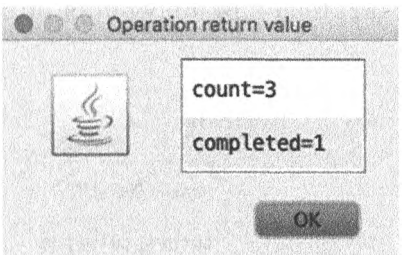
3. Выберите в главной форме вкладку MBeans. Разверните пакет `org.springframework.boot` и каталог `Endpoint`. Вы увидите пункт меню для `Todo-stats`. Можете развернуть его и посмотреть на все операции.



4. Щелкните на пункте `stats`, чтобы увидеть статистику операции MBeans.

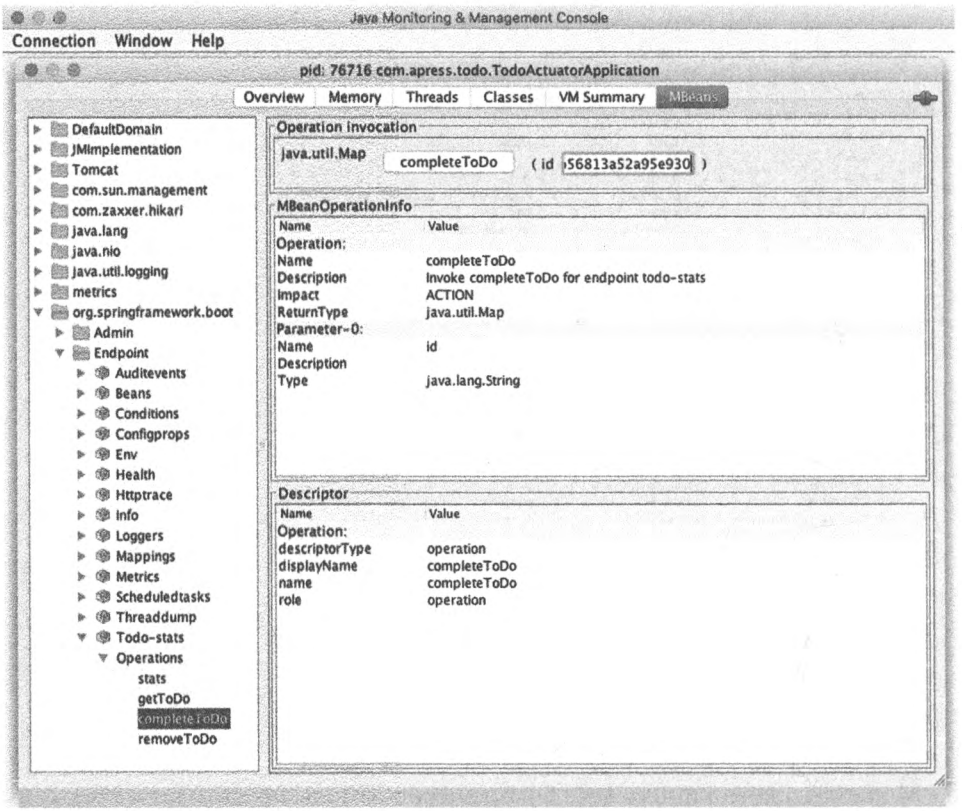


5. Можете нажать кнопку stats (фактически это вызов метода `stats`), и вы получите следующее:

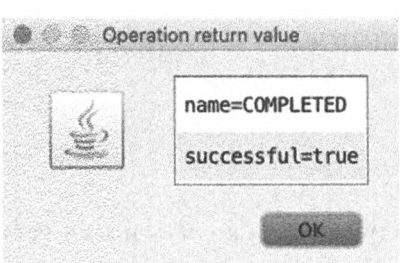


Как видите, вы получили то же, что и при веб-доступе. Можете поэкспериментировать с операцией `completeToDo`.

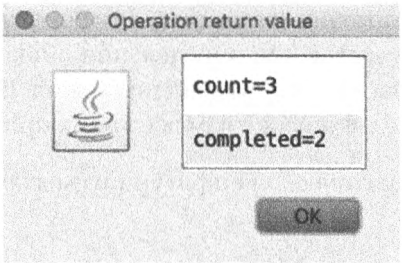
6. Щелкните на операции `completeToDo`. Справа вставьте в поле `ID` значение `ebcf1850563c4de3b56813a52a95e930`, соответствующее незавершенному пока что запланированному делу Buy Movie Tickets.



7. Щелкните на `completeToDo` для получения подтверждения (объекта `Operation`).



8. Если повторить операцию `stats`, вы должны увидеть, что оба дела теперь завершены.



Как видите, использовать JMX с помощью утилиты JConsole очень просто. Теперь вы знаете, как создать пользовательскую конечную точку для получения нужных вам данных.

Конечная точка health Spring Boot Actuator

Сегодня мы стремимся к наблюдаемости систем в том смысле, что хотим внимательно наблюдать за ними, реагируя на любое событие. Я помню, что когда-то давно мониторинг серверов осуществлялся с помощью простого ping, но сейчас этого недостаточно. Нужен мониторинг не только серверов, но и систем, включая их внутренние механизмы. По-прежнему необходимо знать, работает ли система, но если нет, нам требуется больше информации о сути проблемы.

На помощь нам приходит конечная точка `health` Spring Boot Actuator! Конечная точка `/actuator/health` дает информацию о состоянии работающего приложения. В частности, она предоставляет свойство `management.endpoint.health.show-details`, позволяющее отображать при необходимости больше информации обо всей системе. Вот список возможных значений этого свойства.

- `never`. Подробная информация никогда не отображается; поведение по умолчанию.
- `when-authorized`. Подробная информация отображается только для авторизованных пользователей; для настройки ролей служит свойство `management.endpoint.health.roles`.
- `always`. Вся подробная информация отображается для всех пользователей.

Spring Boot Actuator предоставляет интерфейс `HealthIndicator`, собирающий всю информацию о системе; он возвращает содержащий всю эту информацию экземпляр класса `Health`. В Spring Boot Actuator есть несколько готовых индикаторов состояния приложения, автоматически конфигурируемых с помощью интерфейса `HealthAggregator` для определения общего состояния системы. Все это очень напоминает уровни журналирования. Не волнуйтесь и просто запускайте приложение. Я покажу вам все это на примере.

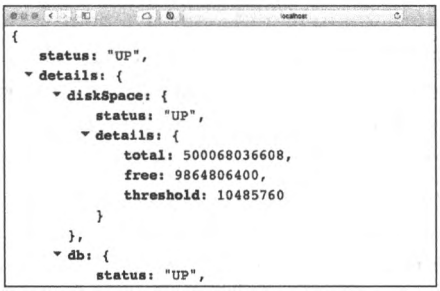
Вот некоторые из автоматически конфигурируемых индикаторов состояния приложения.

- `CassandraHealthIndicator` — проверяет, запущена ли база данных Cassandra.
- `DiskSpaceHealthIndicator` — проверяет, достаточно ли места на диске.
- `RabbitHealthIndicator` — проверяет, запущен ли сервер Rabbit.
- `RedisHealthIndicator` — проверяет, запущен ли сервер Redis.
- `DataSourceHealthIndicator` — проверяет соединение источника данных с базой и может также выполнить тестовый запрос.
- `MongoHealthIndicator` — проверяет, запущен ли сервер MongoDB.
- `MailHealthIndicator` — проверяет, работает ли сервер электронной почты.
- `SolrHealthIndicator` — проверяет, запущен ли сервер Solr.
- `JmsHealthIndicator` — проверяет, запущен ли сервер JMS.
- `ElasticsearchHealthIndicator` — проверяет, работает ли кластер ElasticSearch.
- `Neo4jHealthIndicator` — проверяет, запущен ли сервер Neo4j.
- `InfluxDBHealthIndicator` — проверяет, запущен ли сервер InfluxDB.

Это далеко не все, существует множество других. Все они автоматически настраиваются, достаточно наличия зависимости по пути к классам; другими словами, для их настройки и использования ничего делать не нужно.

Попробуем воспользоваться индикатором состояния приложения.

1. Убедитесь в наличии в файле `application.properties` приложения ToDo свойства `endpoints.web.exposure.include=*`.
2. Добавьте в файл `application.properties` свойство `management.endpoint.health.show-details=always`.
3. Если вы теперь запустите приложение ToDo и обратитесь к конечной точке http://localhost:8080/actuator/health, то должны увидеть следующее.



`DataSourceHealthIndicator` для базы данных H2 и `DiskSpaceHealthIndicator` были настроены автоматически.

4. Добавьте следующую зависимость в файл `pom.xml` (если вы используете Maven):

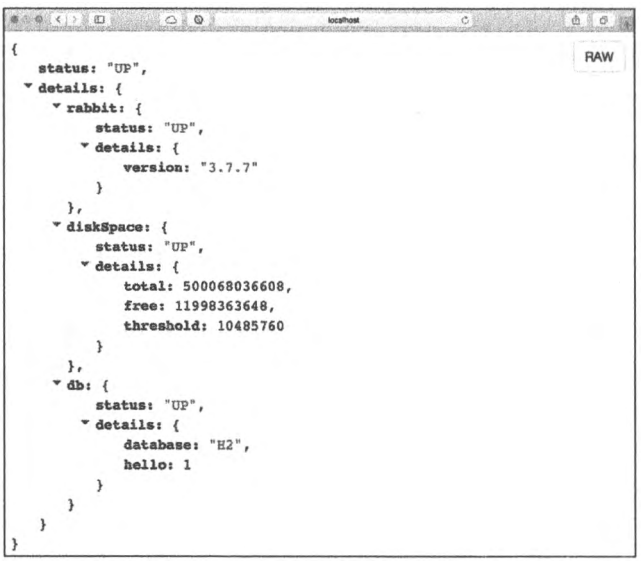
```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

   Если вы используете Gradle, добавьте в файл `build.gradle` следующую зависимость:

```
compile('org.springframework.boot:spring-boot-starter-amqp')
```

5. Да, вы правильно поняли. Мы добавляем зависимости AMQP. Перезапустите приложение и взгляните на конечную точку `/actuator/health`.

Поскольку мы добавили зависимость `spring-boot-starter-amqp`, речь идет о брокере RabbitMQ и у вас имеется Actuator, `RabbitHealthIndicator` автоматически настроен для доступа к локальному хосту (или конкретному брокеру в соответствии с настройками `spring.rabbitmq.*`). Если он работает нормально, вы увидите сообщение об этом. В данном же случае можно видеть в журнале неудачное соединение, а в конечной точке `health` видно, что система не работает. Если у вас есть брокер RabbitMQ (из предыдущей главы), то можете его запустить (с помощью команды `rabbitmq-server`) и обновить страницу конечной точки `health`. Как видите, все работает!



Вот и все. Именно так можно работать с готовыми индикаторами состояния приложения. Просто добавьте соответствующую зависимость — и можете пользоваться!

Создание приложения ToDo с пользовательским индикатором состояния приложения

Пришло время приложению ToDo обзавестись своим пользовательским индикатором состояния приложения. Необходимо реализовать интерфейс `HealthIndicator`, возвращая желаемую информацию о состоянии в экземпляре `Health`.

Создайте класс ToDoHealthCheck для обращения к пути в файловой системе и выполнения проверки его на предмет доступности, а также возможностей чтения и записи (листинг 10.5).

Листинг 10.5[1]. com.apress.todo.actuator.ToDoHealthCheck.java

```
package com.apress.todo.actuator;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.actuate.health.Health;
import org.springframework.stereotype.Component;

import java.io.File;

@Component
public class ToDoHealthCheck implements HealthIndicator {

    private String path;

    public ToDoHealthCheck(@Value("${todo.path:/tmp}")String path){
        this.path = path;
    }

    @Override
    public Health health() {

        try {

            File file = new File(path);
            if(file.exists()){

                if(file.canWrite())
                    return Health.up().build();

                return Health.down().build();

            }else{
                return Health.outOfService().build();
            }
        }catch(Exception ex) {
            return Health.down(ex).build();
        }
    }
}
```

[1] В этом коде не хватает одной нужной операции импорта:

import org.springframework.boot.actuate.health.HealthIndicator

В листинге 10.5 приведен снабженный аннотацией `@Component` класс `ToDoHealthCheck` — реализация интерфейса `HealthIndicator`. Необходимо реализовать метод `health` (обратите внимание, что у класса `Health` есть текучий API, помогающий создавать объект с состоянием приложения). Проанализируйте код и обратите внимание, что значение переменной `path` должно задаваться в свойстве среды, аргументе командной строки или в файле `application.properties`; в противном случае по умолчанию используется путь `/tmp`. Если путь существует, проверяется возможность записи; в случае положительного результата этой проверки отображается состояние `UP`, в противном случае — `DOWN`. Если путь не существует, возвращается состояние `OUT_OF_SERVICE`. В случае какого-либо исключения отображается состояние `DOWN`.

В предыдущем коде требуется свойство `todo.path`. Создадим класс `ToDoProperties` для информации об этом свойстве. Он вам уже знаком (листинг 10.6).

Листинг 10.6. com.apress.todo.config.ToDoProperties.java

```
package com.apress.todo.config;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

@Data
@ConfigurationProperties(prefix = "todo")
public class ToDoProperties {
    private String path;
}
```

Как видите, он очень прост. Если вы помните, для использования снабженного аннотацией `@ConfigurationProperties` класса необходимо вызывать его с помощью аннотации `@EnableConfigurationProperties`. Создадим класс `ToDoConfig` для этой цели (листинг 10.7).

Листинг 10.7. com.apress.todo.config.ToDoConfig.java

```
package com.apress.todo.config;

import org.springframework.boot.context.properties.
  EnableConfigurationProperties;
import org.springframework.context.annotation.Configuration;
```

```
@EnableConfigurationProperties(ToDoProperties.class)
@Configuration
public class ToDoConfig {
}
```

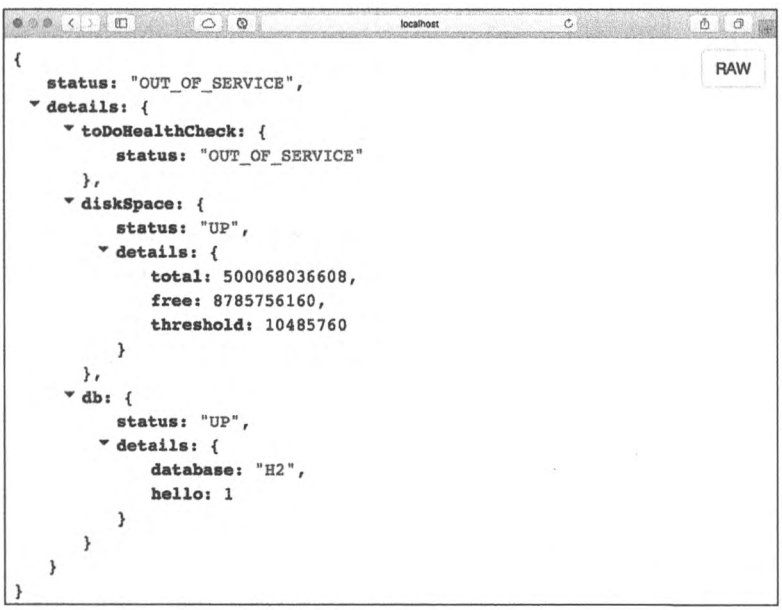
Ничего особенного в этом классе нет. Добавьте в файл `application.properties` новое свойство:

```
todo.path=/tmp/todo
```

Если же вы используете Windows, то можете поэкспериментировать с чем-то вроде:

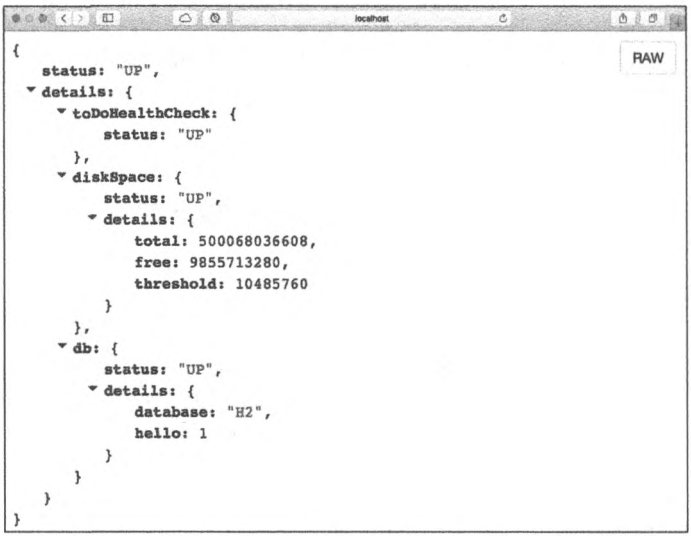
```
todo.path=C:\\tmp\\todo
```

Обратитесь к документации за информацией о том, как правильно задать этот путь. Итак, все готово. Если вы перезапустите приложение ToDo и взглянете на ответ конечной точки `/actuator/health`, то увидите примерно следующее.



В JSON-ответе присутствует ключ `toDoHealthCheck`, отражающий положение вещей.

Далее исправьте проблему, создав доступный для записи каталог `/tmp/todo`, и обновите полученную страницу.



Существует возможность настраивать порядок отображения состояния/серьезности проблемы (например, уровень журналирования) с помощью следующего свойства в файле `application.properties`:

```
management.health.status.order=FATAL, DOWN, OUT_OF_SERVICE, UNKNOWN, UP
```

При использовании конечной точки `health` через HTTP каждому состоянию/серьезности проблемы соответствует свой код состояния HTTP либо можно назначить подобное соответствие.

- `DOWN` — 503.
- `OUT_OF_SERVICE` — 503.
- `UP` — 200.
- `DOWN` — 200.

Свой собственный код состояния можно использовать с помощью свойства:

```
management.health.status.http-mapping.FATAL=503
```

Можно также создавать собственные названия состояний, например `IN_BAD_CONDITION`, с помощью такого вызова:

```
Health.status("IN_BAD_CONDITION").build();
```

Создавать пользовательские индикаторы состояния приложения с помощью Spring Boot Actuator очень просто!

Метрики Spring Boot Actuator

Мониторинг в наши дни необходим для любой системы. Система должна быть наблюдаемой, происходящее должно отслеживаться, как в каждом из приложений по отдельности, так и в системе в целом. Spring Boot Actuator предоставляет основные метрики, интеграцию и автоконфигурацию для Micrometer (http://micrometer.io).

Micrometer — простой фасад для клиентов многих популярных систем мониторинга; другими словами, вы можете один раз написать код мониторинга и использовать его для любых сторонних систем, таких как Prometheus, Netflix Atlas, CloudWatch, Datadog, Graphite, Ganglia, JMX, InfluxDB/Telegraf, New Relic, StatsD, SignalFX и WaveFront (а в скором времени и других).

Напомню, что у Spring Boot Actuator есть конечная точка `/actuator/metrics`. Как вы знаете из предыдущих разделов, после запуска приложения ToDo у вас есть доступ к основным метрикам. А вот что я вам пока что не показывал, так это как создавать пользовательские метрики с помощью Micrometer. Его основная идея — написать код мониторинга один раз и иметь возможность применять любую стороннюю утилиту мониторинга. Spring Boot Actuator и Micrometer обеспечивают доступ выбранной утилиты мониторинга к этим метрикам.

Перейдем непосредственно к реализации кода для Micrometer и к использованию Prometheus и Grafana и продемонстрируем, насколько это просто.

Создание приложения ToDo с Micrometer: Prometheus и Grafana

Реализуем необходимый для Micrometer код и воспользуемся Prometheus и Grafana.

Пока мы видели, что модуль Spring Data REST создает для нас веб-контроллеры MVC (конечные точки REST) после обнаружения всех интерфейсов, расширяющих интерфейс `Repository<T,ID>`. Представьте на минуту, что нам нужно перехватить эти запросы и начать формировать сводный показатель, метрику, отражающую число запросов для конкретной конечной точки REST

и конкретного HTTP-метода. Благодаря этому можно будет найти конечные точки, подходящие для создания микросервиса. Подобный перехватчик будет получать все запросы, включая предназначенные для `/actuator`.

Для этого Spring MVC предоставляет интерфейс `HandlerInterceptor`. Он включает три метода с реализациями по умолчанию, из которых нам нужен только один. Начнем с создания класса `ToDoMetricInterceptor` (листинг 10.8).

Листинг 10.8. com.apress.todo.interceptor.ToDoMetricInterceptor.java

```
package com.apress.todo.interceptor;

import io.micrometer.core.instrument.MeterRegistry;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.servlet.HandlerInterceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ToDoMetricInterceptor implements HandlerInterceptor {

     private static Logger log = LoggerFactory.
         getLogger(ToDoMetricInterceptor.class);

    private MeterRegistry registry;
    private String URI, pathKey, METHOD;

    public ToDoMetricInterceptor(MeterRegistry registry) {
        this.registry = registry;
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
    HttpServletResponse response, Object handler, Exception ex)
    throws Exception {
        URI = request.getRequestURI();
        METHOD = request.getMethod();
        if (!URI.contains("prometheus")){
            log.info(" >> PATH: {}",URI);
            log.info(" >> METHOD: {}", METHOD);
            pathKey = "api_".concat(METHOD.toLowerCase()).
            concat(URI.replaceAll("/","_").toLowerCase());
            this.registry.counter(pathKey).increment();
        }
    }
}
```

В листинге 10.8 показан класс `ToDoMetricInterceptor`, реализующий интерфейс `HandlerInterceptor`. У этого интерфейса есть три метода с реализациями по умолчанию: `preHandle`, `postHandle` и `afterCompletion`. Данный класс реализует только метод `afterCompletion`. В качестве одного из параметров этому методу передается объект `HttpServletRequest`, с помощью которого можно выяснить, к какой конечной точке и с помощью какого HTTP-метода производился запрос.

Этот класс использует экземпляр входящего во фреймворк Micrometer класса `MeterRegistry`. Наша реализация получает путь и метод из экземпляра запроса, после чего наращивает значение `registry` с помощью метода `counter`. `pathKey` очень прост; при запросе `Get` к конечной точке `/toDos` `pathKey` равен `api_get_todos`. При запросе `POST` к конечной точке `/toDos` `pathKey` равен `api_post_todos` и т. д. Так что в случае нескольких запросов к конечной точке `/toDos` значение `registry` наращивается (с помощью этого `pathKey`) и агрегируется с существующим значением.

Теперь необходимо обеспечить автоматическое обнаружение и настройку компонента `ToDoMetricInterceptor` модулем Spring MVC. Откройте класс `ToDoConfig` и добавьте туда компонент `MappedInterceptor` (см. версию 2 в листинге 10.9).

Листинг 10.9. com.apress.todo.config.ToDoConfig.java — версия 2

```
package com.apress.todo.config;

import com.apress.todo.interceptor.ToDoMetricInterceptor;
import io.micrometer.core.instrument.MeterRegistry;
import org.springframework.boot.context.properties.
  EnableConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.handler.MappedInterceptor;

@EnableConfigurationProperties(ToDoProperties.class)
@Configuration
public class ToDoConfig {
   @Bean
   public MappedInterceptor metricInterceptor(MeterRegistry registry) {
       return new MappedInterceptor(new String[]{"/**"},
                       new ToDoMetricInterceptor(registry));
   }
}
```

В листинге 10.9 приведена новая версия класса `ToDoConfig` с компонентом `MappedInterceptor`, использующим для каждого из запросов `ToDoMetricInterceptor` с помощью сопоставления с шаблоном `"/**"`.

Далее добавим две зависимости для экспорта данных, в JMX и Prometheus. Если вы используете Maven, добавьте следующие зависимости в файл `pom.xml`:

```
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-jmx</artifactId>
</dependency>
<dependency>
    <groupId>io.micrometer</groupId>
    <artifactId>micrometer-registry-prometheus</artifactId>
</dependency>
```

Если вы применяете Gradle, добавьте следующие зависимости в файл `build.gradle`:

```
compile('io.micrometer:micrometer-registry-jmx')
compile('io.micrometer:micrometer-registry-prometheus')
```

Spring Boot Actuator автоматически настраивает и регистрирует все `registry` Micrometer, в данном случае JMX и Prometheus. Для Prometheus актуатор настраивает конечную точку `/actuator/prometheus`.

Предварительные требования: использование Docker

Прежде чем проверять работу приложения ToDo с помощью метрик, необходимо установить Docker (постарайтесь установить последнюю версию).

- Установите Docker CE (Community Edition) из https://docs.docker.com/install/.
- Установите Docker Compose из https://docs.docker.com/compose/install/.

Почему я выбрал Docker? Потому что это простейший способ установки всего необходимого. И мы будем использовать его еще в следующих главах. Docker Compose упрощает установку Prometheus и Grafana за счет применения внутренней сети Docker, позволяющей использовать DNS-имена.

docker-compose.yml

Ниже приведен файл `docker-compose.yml`, используемый для запуска Prometheus и Grafana.

```
version: '3.1'

networks:
  micrometer:

services:

  prometheus:
    image: prom/prometheus
    volumes:
      - ./prometheus/:/etc/prometheus/
    command:
      - '--config.file=/etc/prometheus/prometheus.yml'
      - '--storage.tsdb.path=/prometheus'
      - '--web.console.libraries=/usr/share/prometheus/console_libraries'
      - '--web.console.templates=/usr/share/prometheus/consoles'
    ports:
      - 9090:9090
    networks:
      - micrometer
    restart: always

  grafana:
    image: grafana/grafana
    user: "104"
    depends_on:
      - prometheus
    volumes:
      - ./grafana/:/etc/grafana/
    ports:
      - 3000:3000
    networks:
      - micrometer
    restart: always
```

Можете создать этот файл с помощью любого редактора. Помните, это файл, в котором для отступов не используется табуляция. Вам нужно будет создать два каталога: `prometheus` и `grafana`. В каждом из них будет по одному файлу.

В каталоге `prometheus` должен быть файл `prometheus.yml` со следующим содержимым:

```
global:
  scrape_interval:     5s
  evaluation_interval: 5s

scrape_configs:
  - job_name: 'todo-app'
```

```
metrics_path: '/actuator/prometheus'
scrape_interval: 5s
static_configs:
  - targets: ['host.docker.internal:8080']
```

В этом файле следует обратить внимание на `metrics_path` и целевые ключи. Обнаруживая зависимость `micrometer-registry-prometheus`, Spring Boot Actuator автоматически настраивает конечную точку `/actuator/prometheus`. Необходимо задать это значение для `metrics_path`. Еще одно важное значение — ключ `targets`. Prometheus опрашивает конечную точку `actuator/prometheus` каждые пять секунд. Ему необходимо знать, где она расположена (он использует для этого доменное имя `host.docker.internal`). Эта часть Docker выполняет поиск хоста (запущенного приложения `localhost:8080/todo-actuator`).

Каталог `grafana` содержит пустой файл `grafana.ini`. Чтобы утилита Grafana точно использовала значения по умолчанию, структура каталогов должна быть следующей.

```
.
├── docker-compose.yml
├── grafana
│   └── grafana.ini
└── prometheus
    └── prometheus.yml

2 directories, 3 files
```

Запуск метрик приложения ToDo

Пришло время приступить к тестированию и настройке Grafana для получения удобных метрик работы приложения. Запустите приложение ToDo. Посмотрите в журнал и убедитесь в наличии конечной точки `/actuator/prometheus`.

Откройте терминал, перейдите в каталог с файлом `docker-compose.yml` и выполните следующую команду:

```
$ docker-compose up
```

Эта команда запускает механизм `docker-compose`, скачивающий образы Docker и запускающий их. Убедитесь, что приложение Prometheus работает, открыв браузер и перейдя по адресу http://localhost:9090/targets.

Это означает успешную загрузку конфигурации `prometheus.yml`. Другими словами, Prometheus сейчас опрашивает конечную точку http://localhost:8080/actuator/prometheus.

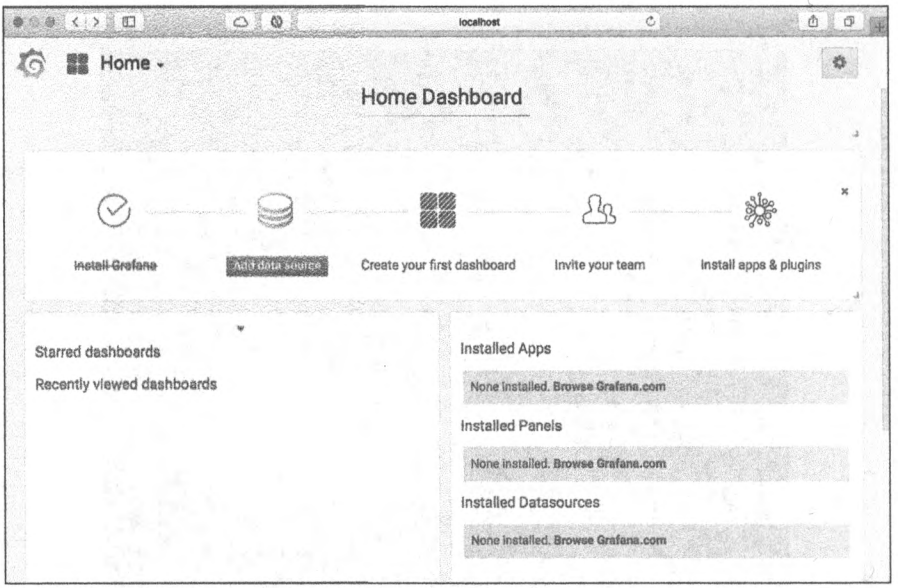
Теперь настроим Grafana.

1. Откройте браузер и перейдите по адресу http://localhost:3000.

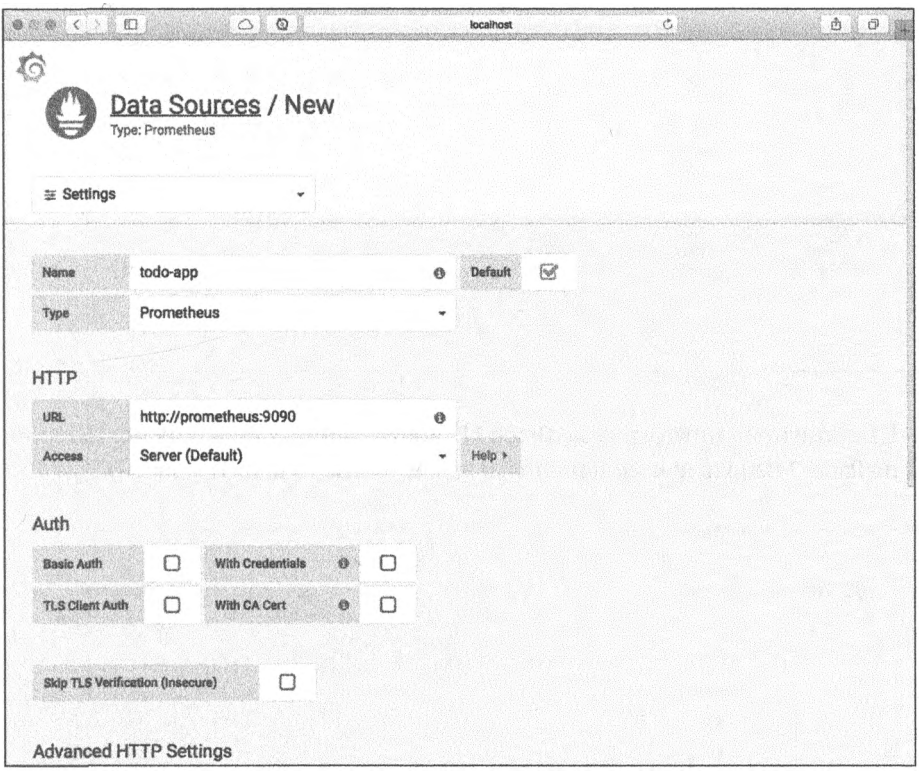
   Можете использовать учетные данные `admin`/`admin`.

2. Можете нажать кнопку **Skip** (Пропустить) для пропуска следующей экранной формы.

3. Щелкните на пиктограмме **Add Data Source** (Добавить источник данных).

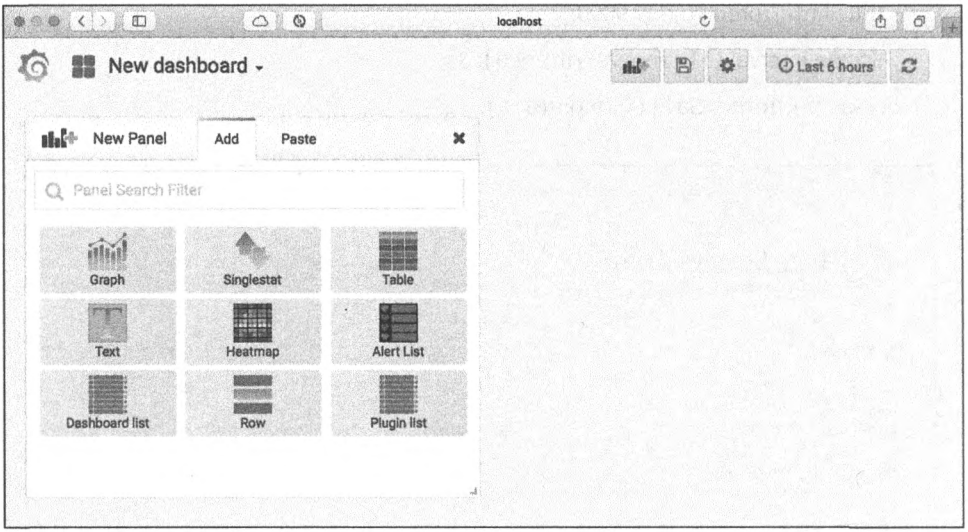


4. Заполните требуемые поля. Главные из них:
   - Name (Название): `todo-app`;
   - Type (Тип): `Prometheus`;
   - URL: http://prometheus:9090;
   - Scrape interval (Интервал опроса): `3s`.
5. Нажмите кнопку Save (Сохранить).

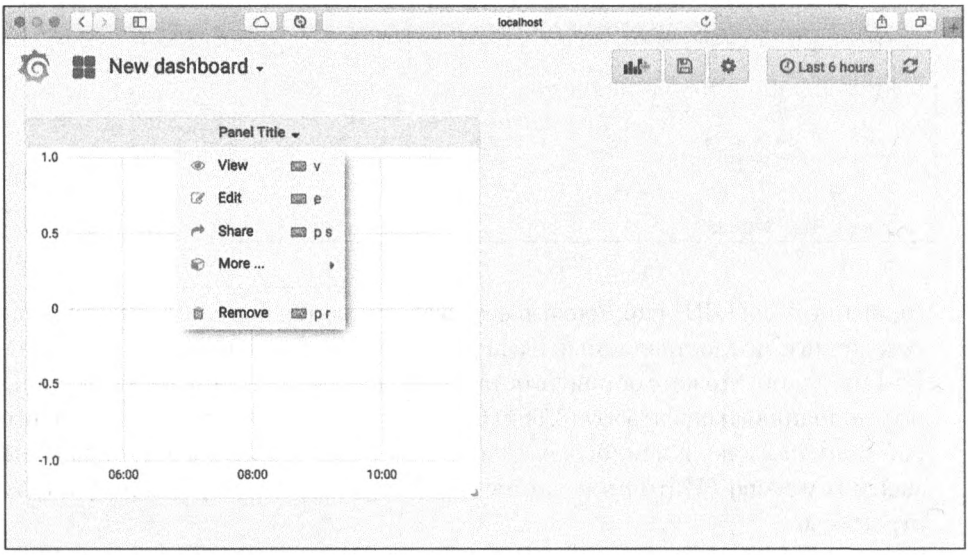


Значение поля URL, http://prometheus:9090, ссылается на сервис docker-compose, точнее, предоставляемый им внутренний DNS, так что указывать localhost не нужно. Можете оставить остальные значения такими, какие указаны по умолчанию, и нажать Save & Test (Сохранить и протестировать). Если все работает так, как ожидалось, вы увидите зеленый баннер с надписью Data source is working (Источник данных функционирует нормально) внизу страницы.

6. Можете перейти на домашнюю страницу, вернувшись обратно или перейдя в браузере по адресу http://localhost:3000. Теперь можно нажать кнопку New Dashboard (Новая инструментальная панель) на домашней странице.

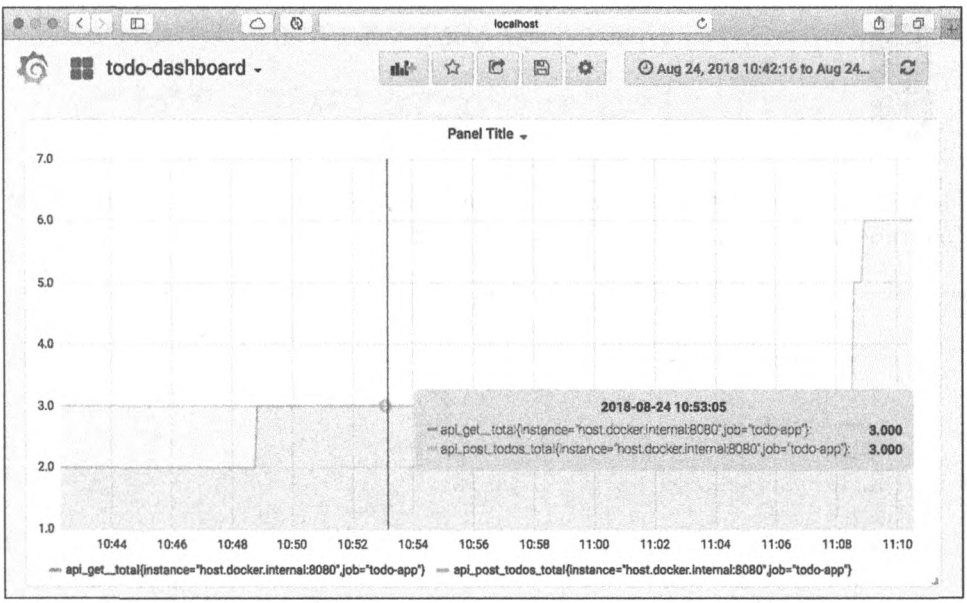


7. Щелкните на пиктограмме Graph (График), и появится панель. Щелкните на Panel Title (Название панели), а затем на Edit (Редактировать).



8. Настраиваем два сгенерированных Micrometer и Spring Boot Actuator в качестве метрик для движка Prometheus запроса, `api_get_total` и `api_post_todos_total`.

9. Выполните эти запросы `GET` (несколько раз) и `POST` к конечной точке `/toDos`. Вы увидите нечто напоминающее следующий рисунок.



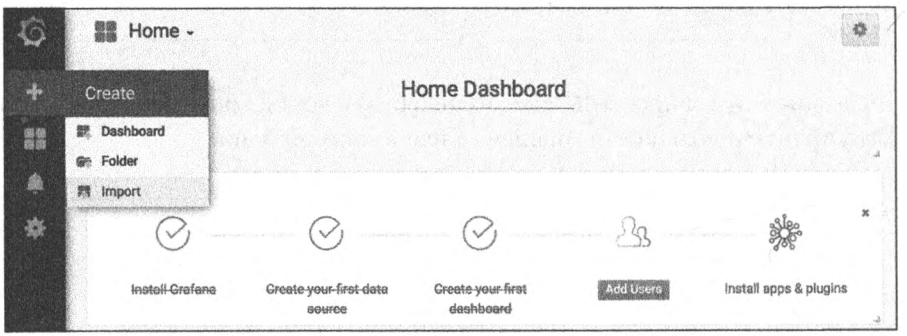
Поздравляю! Вы создали пользовательские метрики с помощью Micrometer, Prometheus и Grafana.

Получение общей статистики для Spring Boot с помощью Grafana

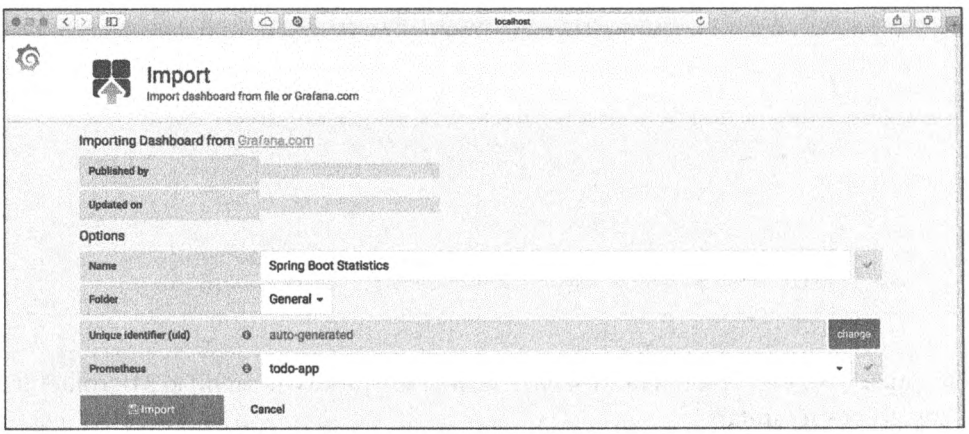
Я обнаружил очень удобную конфигурацию для Grafana, с помощью которой можно воспользоваться всеми предоставляемыми Spring Boot Actuator метриками. Эту конфигурацию можно импортировать в Grafana.

Скачайте эту конфигурацию по адресу https://grafana.com/dashboards/6756. Нужный вам файл называется `spring-boot-statistics_rev2.json`.

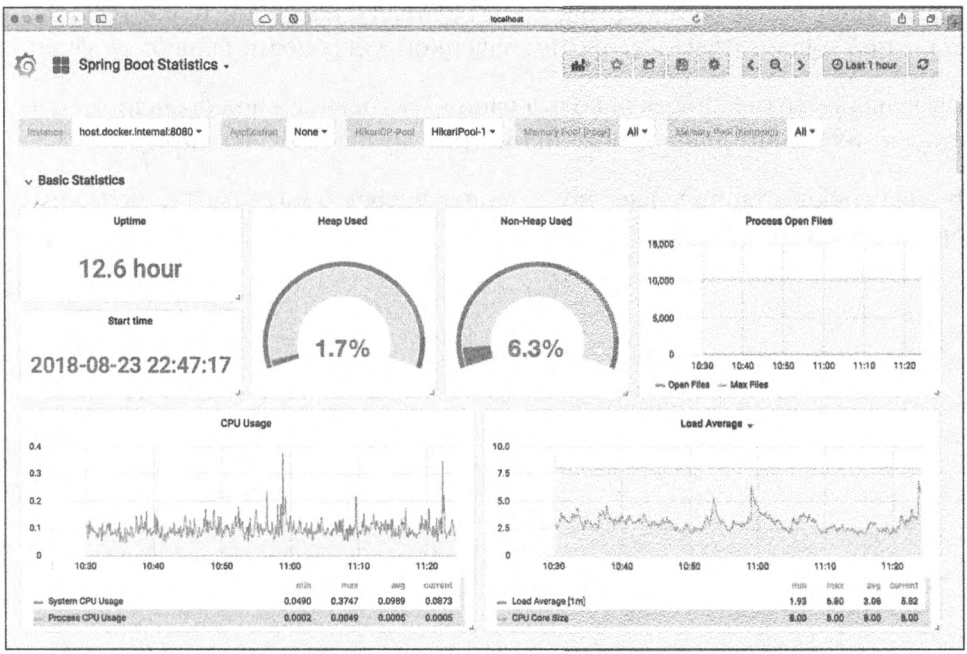
Щелкните на пиктограмме Grafana в левом углу домашней страницы Grafana (http://localhost:3000). В результате откроется боковая панель. Щелкните на символе + и выберите пункт Import (Импортировать).



Не трогайте значения по умолчанию, но в поле Prometheus выберите todo-app (настроенный нами ранее источник данных).



Щелкните на пункте Import (Импортировать) — и вуаля! Получите полную инструментальную панель со всеми метриками и возможностями мониторинга Spring Boot Actuator!



Изучите внимательно все графики. Все данные поступают из конечной точки `/actuator/prometheus`.

Можете теперь остановить выполнение `docker-compose`, выполнив следующую команду в отдельном окне терминала:

```
$ docker-compose down
```

ПРИМЕЧАНИЕ

Программное решение для этого раздела можно найти в прилагаемом к книге исходном коде на веб-сайте Apress или в GitHub по адресу https://github.com/Apress/pro-spring-boot-2 либо в моем личном репозитории по адресу https://github.com/felipeg48/pro-spring-boot-2nd.

Резюме

В этой главе мы обсудили Spring Boot Actuator, включая его конечные точки и возможности их настройки. С помощью модуля Actuator можно производить мониторинг приложения Spring Boot, начиная с конечной точки `/health` до конечной точки `/httptrace`, предназначенной для более детальной отладки.

Вы узнали о возможности использования Micrometer и подключения любых сторонних утилит для применения метрик Spring Boot Actuator.

В следующей главе мы сделаем еще шаг вперед и научимся использовать Spring Integration и Spring Cloud Stream.

11 Создание приложений Spring Integration и Spring Cloud Stream

В этой главе я покажу вам один из лучших интеграционных фреймворков для Java: основанный на фреймворке Spring проект Spring Integration. Я также познакомлю вас с основанным на Spring Integration проектом Spring Cloud Stream. С его помощью можно создавать надежные и масштабируемые событийно-управляемые микросервисы, подключенные к системам разделяемого обмена сообщениями, — все благодаря Spring Boot.

Если смотреть на разработку программного обеспечения и бизнес-потребности с точки зрения разработчика или архитектора, речь всегда идет об интеграции компонентов и систем, либо внутренних по отношению к нашей архитектуре, либо внешних, и выяснении, какие варианты обеспечивают полную функциональность, высокую доступность и легко поддаются сопровождению и расширению.

Вот основные сценарии использования, с которыми обычно сталкиваются разработчики и архитекторы.

- Создание системы, обеспечивающей надежную передачу или анализ файлов. Большинству приложений необходимо читать информацию из файлов, а затем ее обрабатывать, так что файловые системы с сохранением и чтением данных должны быть надежными, а также уметь обрабатывать файлы различного размера.
- Возможность использовать данные в разделяемой среде, в которой возможность выполнения операций и доступ к одной базе данных или таблице необходим нескольким клиентам (системам или пользователям), с учетом вопросов несогласованности, дублирования данных и пр.

- Удаленный доступ к различным системам, начиная от выполнения удаленных процедур и до отправки больших объемов информации. Причем желательно в режиме реального времени и асинхронно. Плюс необходимость получения ответа как можно быстрее, с учетом того, что удаленная система должна в любой момент оставаться доступной; другими словами, необходимы отказоустойчивость и высокая доступность.
- Обмен сообщениями — от простейших внутренних вызовов до отправки миллиардов сообщений в секунду удаленным брокерам. Обычно обмен сообщениями производится асинхронно, так что необходимы параллелизм, многопоточность, скорость (низкое значение сетевой задержки), высокая доступность, отказоустойчивость и т. д.

Как же решить/реализовать все эти сценарии использования? Почти 15 лет назад специалисты по ПО Грегор Хоп (Gregor Hohpe) и Бобби Вульф (Bobby Woolf) написали книгу *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions* (Addison-Wesley, 2003)[1]. В ней описаны все паттерны обмена сообщениями, необходимые для реализации вышеупомянутых сценариев использования. Она объясняет понятными словами, как системы взаимодействуют и работают и как создать надежную систему интеграции с архитектурой приложения на основе объектно-ориентированного проектирования, направленного на обработку сообщений.

В следующих разделах я покажу вам часть из этих паттернов с помощью проекта Spring Integration из фреймворка Spring.

Азбука Spring Integration

Spring Integration — простая модель реализации корпоративных решений интеграции. С ее помощью можно сделать приложение Spring Boot асинхронным и ориентированным на обработку сообщений. Spring Integration благодаря реализации всех корпоративных паттернов интеграции позволяет создавать корпоративные, надежные, переносимые интегрированные решения.

---

[1] *Хоп Г., Вульф Б.* Шаблоны интеграции корпоративных приложений. Проектирование, создание и развертывание решений, основанных на обмене сообщениями. — М.: Вильямс, 2019.

Проект Spring Integration обеспечивает модульную организацию и тестируемость компонентов за счет их слабой связанности. Он помогает обеспечить разделение ответственности между бизнес-логикой и логикой интеграции.

Основные компоненты Spring Integration таковы.

- *Сообщение* (message). Универсальная обертка для любого Java-объекта. Состоит из заголовков и содержимого. Заголовки обычно содержат важную информацию, например идентификатор, метку даты/времени, идентификатор корреляции и обратный адрес, и конечно, вы можете добавлять свои собственные поля. Содержимое может представлять собой данные произвольного типа, от байтовых массивов до пользовательских объектов. Описание заголовков и содержимого можно найти в модуле `spring-messaging` в пакете `org.springframework.messaging`.

```
public interface Message<T> {
      T getPayload();
      MessageHeaders getHeaders();
}
```

  Как видите, ничего необычного в этом определении нет.

- *Канал передачи сообщений* (message channel). Архитектура, состоящая из программных каналов и фильтров, очень похожа на команды операционной системы Unix. Для ее использования необходимы генераторы и потребители: генератор отправляет сообщение в канал передачи сообщений, а потребитель его получает (рис. 11.1).

Рис. 11.1. Канал передачи сообщений

Такой канал передачи сообщений следует паттернам обмена сообщениями, например, паттерну «точка — точка» и модели «публикация/подписка». В Spring Integration есть несколько каналов передачи сообщений, в частности каналы передачи сообщений с опросом (благодаря которым возможна буферизация сообщений в очереди) или каналы передачи сообщений с подпиской, для потребителей.

- *Конечная точка сообщений* (Message endpoint). Фильтр, соединяющий код приложения с фреймворком обмена сообщениями. Большинство этих конечных точек входят в число реализаций из *Enterprise Integration Patterns*.
- *Фильтр* (Filter). Фильтр сообщений определяет, когда сообщение следует передавать в выходной канал.
- *Модификатор* (Transformer). Модификатор сообщений модифицирует содержимое или структуру сообщения и передает его в выходной канал.
- *Маршрутизатор* (Router). Маршрутизатор сообщений определяет на основе заданных правил, что делать и куда отправлять сообщение. Эти правила могут содержаться в заголовках или в содержимом сообщения. Существует множество возможных паттернов для маршрутизаторов сообщений. Я покажу вам по крайней мере один из них.
- *Разделитель* (Splitter). Разделитель сообщений получает сообщение (по входному каналу), разбивает его и возвращает несколько новых сообщений (по выходному каналу).
- *Активатор сервиса* (Service activator). Конечная точка, играющая роль сервиса: получающая (по входному каналу) и обрабатывающая сообщение. Может завершать поток интеграции, возвращать то же сообщение или возвращать совершенно новое (по выходному каналу).
- *Агрегатор* (Aggregator). Эта конечная точка сообщений получает несколько сообщений (по входному каналу), объединяет их в одно новое сообщение (на основе стратегии выпуска) и выдает его наружу (по выходному каналу).
- *Адаптеры каналов* (Channel adapters). Конечная точка, соединяющая канал передачи сообщений с другими системами или транспортными протоколами. Spring Integration предоставляет как входные, так и выходные адаптеры. На случай, когда требуется ответ, в нем есть адаптер-шлюз. Они используются чаще всего. Почему? Если вашему приложению нужно подключиться к RabbitMQ, JMS, FTP, файловой системе, HTTP или любой другой технологии, программировать клиент не нужно, в Spring Integration всегда найдется адаптер для подключения к ней.

Описание Spring Integration, паттернов сообщений, каналов передачи сообщений, адаптеров и всего прочего заняло бы целую книгу, так что если эта тема вам интересна — рекомендую книгу Марка Люя (Dr. Mark Lui) *Pro Spring Integration* (Apress, 2011).

В следующем разделе вас ждет достаточное количество компонентов и паттернов, чтобы начать работу со Spring Integration.

Программирование Spring Integration

В случае Spring Integration есть несколько способов настройки всех компонентов (сообщений, каналов передачи сообщений и конечных точек сообщений): XML, классы JavaConfig, аннотации и новый DSL интеграции.

Создание приложения ToDo с помощью Spring Integration

Начнем с уже хорошо нам знакомого приложения ToDo и сразу же воспользуемся Spring Integration. Можете создавать приложение с нуля или просто следить за кодом в следующих разделах, чтобы узнать, что нужно делать. Если начинаете с нуля, то перейдите в Spring Initializr (https://start.spring.io) и внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-integration`.
- Name (Название): `todo-integration`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Spring Integration`, `Lombok`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами интегрированную среду разработки (рис. 11.2).

Как вы видите по зависимостям, теперь мы используем Spring Integration. Можете переиспользовать/скопировать класс `ToDo` (листинг 11.1).

Листинг 11.1. com.apress.todo.domain.ToDo.java

```
package com.apress.todo.domain;

import lombok.Data;
import java.time.LocalDateTime;
import java.util.UUID;

@Data
public class ToDo {

    private String id;
    private String description;
    private LocalDateTime created;
    private LocalDateTime modified;
    private boolean completed;
```

```
    public ToDo(){
        this.id = UUID.randomUUID().toString();
        this.created = LocalDateTime.now();
        this.modified = LocalDateTime.now();
    }

    public ToDo(String description){
        this();
        this.description = description;
    }

    public ToDo(String description,boolean completed){
        this(description);
        this.completed = completed;
    }
}
```

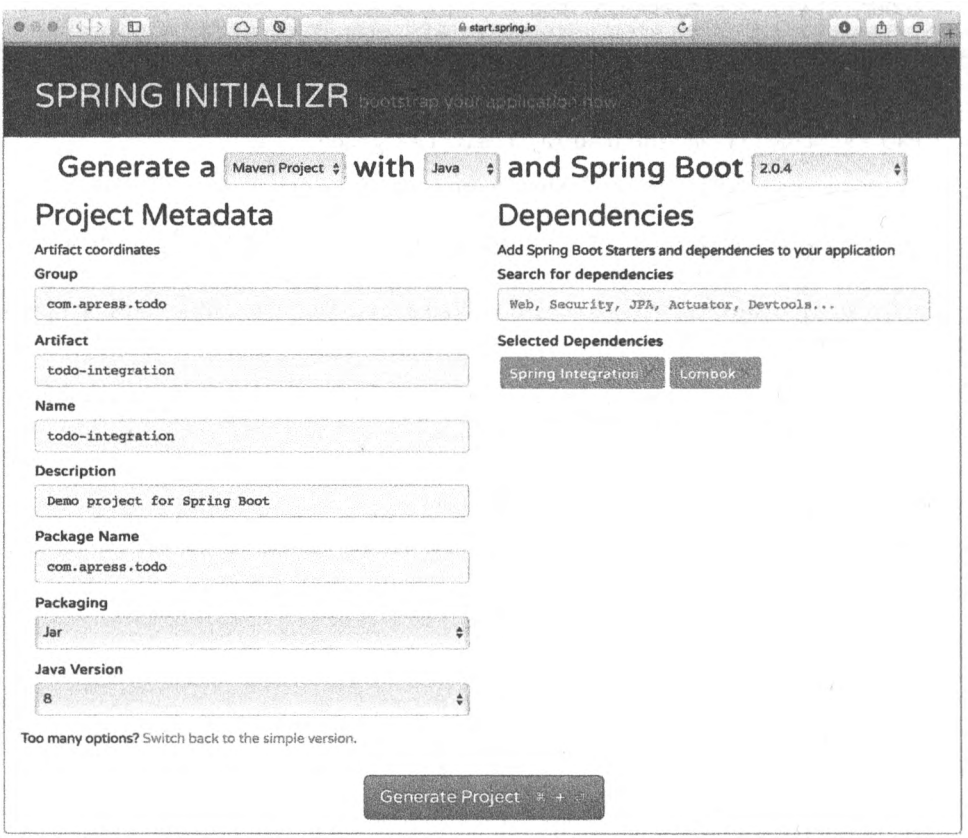


Рис. 11.2. Spring Initializr

В листинге 11.1 приведен хорошо вам знакомый класс `ToDo`. Ничего нового в нем нет. Создадим класс `ToDoIntegration`, включающий первый наш поток интеграции Spring на основе DSL (листинг 11.2).

Листинг 11.2. com.apress.todo.integration.ToDoIntegration.java

```
package com.apress.todo.integration;

import com.apress.todo.domain.ToDo;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.channel.DirectChannel;
import org.springframework.integration.config.EnableIntegration;
import org.springframework.integration.dsl.IntegrationFlow;
import org.springframework.integration.dsl.IntegrationFlows;
import org.springframework.integration.dsl.channel.MessageChannels;

@EnableIntegration
@Configuration
public class ToDoIntegration {

    @Bean
    public DirectChannel input(){
        return MessageChannels.direct().get();
    }

    @Bean
    public IntegrationFlow simpleFlow(){
        return IntegrationFlows
              .from(input())
              .filter(ToDo.class, ToDo::isCompleted)
              .transform(ToDo.class,
                  toDo -> toDo.getDescription().toUpperCase())
              .handle(System.out::println)
                   .get();
     }
}
```

В листинге 11.2 приведен простой пример, получающий сообщение из входного канала (экземпляр `ToDo`), фильтрующий его по признаку завершенности запланированного дела и модифицирующий сообщение (путем приведения описания к верхнему регистру) и выведения в консоль. Все это называется *потоком интеграции* (integration flow). Заглянем внутрь него.

- Интерфейс `IntegrationFlow`. Обеспечивает видимость DSL как компонента (необходима аннотация `@Bean`). Этот интерфейс имеет стандартную реализацию фабрики `IntegrationFlowBuilder`, которая определяет поток

интеграции. Он регистрирует все компоненты, такие как каналы передачи сообщений, конечные точки и т. д.

- Класс `IntegrationFlows`. Этот класс предоставляет текучий API для построения потока интеграции. В него можно легко включать конечные точки для модификации, фильтрации, обработки, разбиения, маршрутизации и сопряжения. В качестве аргументов для этих конечных точек можно использовать любые лямбда-выражения Java 8 (и более поздних версий).
- `from`. Перегруженный метод, в который обычно передается источник сообщения; в данном случае мы вызываем метод `input`, возвращающий экземпляр `DirectChannel` через текучий API `MessageChannels`.
- `filter`. Этот перегруженный метод заполняет `MessageFilter`. `MessageFilter` передает выполнение `MessageSelector`, который, если селектор принимает сообщение, отправляет его в выходной канал фильтра.
- `transform`. Метод может принимать на входе лямбда-выражение, но фактически получает экземпляр `GenericTransformer<S,T>`, где `S` представляет собой источник, а `T` — тип, к которому производится модификация. Здесь можно использовать готовые модификаторы, например `ObjectToJsonTransformer`, `FileToStringTransformer` и т. д. В нашем примере роль источника играет тип класса (`ToDo`) и выполняется лямбда-выражение, в данном случае получение описания запланированного дела и модификация его к верхнему регистру.
- `handle`. Перегруженный метод для заполнения `ServiceActivatingHandler`. Обычно можно использовать POJO, что позволяет получить сообщение и либо вернуть новое сообщение, либо инициировать дополнительный вызов. Это очень полезная конечная точка, которая еще встретится нам в этой и следующей главах в качестве конечной точки активатора сервиса.
- `@EnableIntegration`. Это новая для нас аннотация настраивает все необходимые для нашего потока компоненты Spring Integration. Она регистрирует различные встроенные компоненты, например `errorChannel`, `LoggingHandler`, `taskScheduler` и др., дополняющие наш поток интеграции. Эта аннотация необходима при использовании Java-конфигураций, аннотаций и DSL в приложениях Spring Boot.

Пусть вас не слишком беспокоят отличия от, возможно, созданных вами в прошлом интегрированных решений. Вы привыкнете к этой схеме по мере изучения приведенных далее примеров, и окажется, что она даже проще.

Создадим класс `ToDoConfig`, в котором производится отправка объекта `ToDo` через входной канал (листинг 11.3).

Листинг 11.3. com.apress.todo.config.ToDoConfig.java

```
package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import org.springframework.boot.ApplicationRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.support.MessageBuilder;

@Configuration
public class ToDoConfig {

    @Bean
    public ApplicationRunner runner(MessageChannel input){
        return args -> {
            input.send(
            MessageBuilder
                  .withPayload(new ToDo("buy milk today",true))
                  .build());
        };
    }
}
```

В листинге 11.3 приведен компонент `ApplicationRunner`, выполняемый при запуске приложения (обратите внимание на внедрение компонента `MessageChannel`, объявленного в классе `ToDoIntegration`). В этом методе используется класс `MessageBuilder`, предоставляющий текучий API для создания сообщений. В данном случае для создания нового экземпляра `ToDo`, помеченного как завершенное дело, применяется метод `withPayload` данного класса.

Пришло время запустить наше приложение. Если вы его запустите, то увидите в консоли примерно следующее:

```
...
INFO 39319 - [main] o.s.i.e.EventDrivenConsumer: started simpleFlow.org.
springframework.integration.config.ConsumerEndpointFactoryBean#2
INFO 39319 - [main] c.a.todo.TodoIntegrationApplication      : Started
TodoIntegrationApplication in 0.998 seconds (JVM running for 1.422)
 GenericMessage [payload=BUY MILK TODAY, headers={id=c245b7a3-3191-641b-7ad8-
 1f6eb950f62e, timestamp=1535772540543}]
...
```

Не забывайте, что сообщения состоят из *заголовков* и *содержимого*, поэтому мы получаем объект класса `GenericMessage` с итоговым содержимым `BUY MILK TODAY` и заголовками, включающими идентификатор и метку даты/времени. Это результат применения фильтра и модификации сообщения.

Использование XML

Преобразуем классы так, чтобы использовать другой тип конфигурации, XML, и посмотрим, как в этом случае настроить поток интеграции. Создайте файл `src/main/resources/META-INF/spring/integration/todo-context.xml` (листинг 11.4).

Листинг 11.4. src/main/resources/META-INF/spring/integration/todo-context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:int="http://www.springframework.org/schema/integration"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/integration http://www.
           springframework.org/schema/integration/spring-integration.xsd">

    <int:channel id="input" />
    <int:filter input-channel="input"
                expression="payload.isCompleted()"
                output-channel="filter" />
    <int:channel id="filter" />
    <int:transformer input-channel="filter"
                     expression="payload.getDescription().toUpperCase()"
                     output-channel="log" />
    <int:channel id="log" />
    <int:logging-channel-adapter channel="log" />

</beans>
```

В листинге 11.4 приведена XML-версия конфигурации для потока интеграции `ToDo`. Мне кажется, что она очень проста. Если вы работаете с IDE STS, то можете воспользоваться перетаскиваемой панелью для потоков Spring Integration (схемой интеграции) или сгенерировать схему с помощью IDEA IntelliJ (рис. 11.3).

На рис. 11.3 показана панель для схемы интеграции, в которой можно создавать потоки интеграции визуально, с помощью перетаскиваемых компонентов. Эта возможность есть только в IDE STS. IDEA IntelliJ генерирует схему на основе XML (по щелчку правой кнопкой мыши).

Как вы видите, на рис. 11.3 есть каналы, маршрутизация, модификация, конечные точки и т. д. Рисунок 11.3 фактически представляет собой трансляцию XML; другими словами, можно начать с использования XML и при переходе на схему интеграции посмотреть, что есть на текущий момент, и, наоборот,

можно воспользоваться этой возможностью и переключиться на исходный код, получив XML. Потоки интеграции — потрясающая возможность, правда?

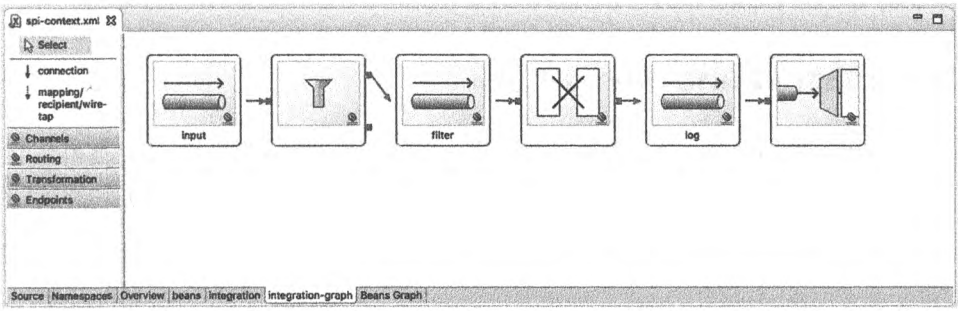


Рис. 11.3. Панель для схемы интеграции Spring из STS

Для запуска этого примера необходимо закомментировать все объявления компонентов из класса `ToDoIntegration`. Далее нам понадобится аннотация `@ImportResource`, чтобы указать Spring Boot, где находится созданная нами XML-конфигурация. В итоге должно получиться нечто похожее на листинг 11.5.

Листинг 11.5. com.apress.todo.integration.ToDoIntregration.java — версия 2

```
package com.apress.todo.integration;

import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.ImportResource;
import org.springframework.integration.config.EnableIntegration;

@ImportResource("META-INF/spring/integration/todo-context.xml")
@EnableIntegration
@Configuration
public class ToDoIntegration {
}
```

В листинге 11.5 приведена новая версия класса `ToDoIntegration` (практически не содержащая кода). Мы добавили аннотацию `@ImportResource`. Она извещает Spring Boot о требующем обработки файле конфигурации. Если запустить приложение, вы увидите следующий вывод:

```
...
INFO 43402 - [main] o.s.i.channel.PublishSubscribeChannel : Channel
'application.errorChannel' has 1 subscriber(s).
2018-09-01 07:23:20.668  INFO 43402 --- [           main] o.s.i.endpoint.
EventDrivenConsumer       : started _org.springframework.integration.
                            errorLogger
```

```
INFO 43402 - [main] c.a.todo.TodoIntegrationApplication      : Started
TodoIntegrationApplication in 1.218 seconds (JVM running for 1.653)
INFO 43402 - [main] o.s.integration.handler.LoggingHandler   : BUY MILK TODAY
...
```

Использование аннотаций

В Spring Integration есть специальные аннотации для интеграции, упрощающие использование классов POJO (простых Java-объектов в старом стиле), благодаря чему можно добавить в поток интеграции дополнительную бизнес-логику и немного улучшить контроль над ним.

Модифицируйте класс `ToDoIntegration`, чтобы он выглядел так, как показано в листинге 11.6.

Листинг 11.6. com.apress.todo.integration.ToDoIntregration.java —версия 3

```
package com.apress.todo.integration;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.annotation.Filter;
import org.springframework.integration.annotation.MessageEndpoint;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.annotation.Transformer;
import org.springframework.integration.channel.DirectChannel;
import org.springframework.integration.config.EnableIntegration;
import org.springframework.integration.dsl.channel.MessageChannels;
import org.springframework.messaging.MessageChannel;

@EnableIntegration
@Configuration
public class ToDoIntegration {

    @Bean
    public MessageChannel input(){
        return new DirectChannel();
    }

    @Bean
    public MessageChannel toTransform(){
        return new DirectChannel();
    }
```

```
    @Bean
    public MessageChannel toLog(){
        return new DirectChannel();
    }

    @MessageEndpoint
    class SimpleFilter {
        @Filter(inputChannel="input"
                        ,outputChannel="toTransform")
        public boolean process(ToDo message){
            return message.isCompleted();
        }
    }

    @MessageEndpoint
    class SimpleTransformer{
        @Transformer(inputChannel="toTransform"
                        ,outputChannel="toLog")
        public String process(ToDo message){
            return message.getDescription().toUpperCase();
        }
    }

    @MessageEndpoint
    class SimpleServiceActivator{
        Logger log = LoggerFactory
                  .getLogger(SimpleServiceActivator.class);
        @ServiceActivator(inputChannel="toLog")
        public void process(String message){
            log.info(message);
        }
    }
}
```

В листинге 11.6 приведен тот же поток интеграции, что и раньше, но теперь с использованием аннотаций интеграции. Обратите также внимание на внутренние вспомогательные классы. Рассмотрим этот код подробнее.

- `MessageChannel` — интерфейс, в котором описаны методы отправки сообщений.
- `DirectChannel` — канал передачи сообщений, вызывающий одного подписчика для каждого отправленного сообщения. Обычно используется, когда очередь сообщений не нужна.
- `@MessageEndpoint` — удобная аннотация, указывающая, что класс является конечной точкой.

- `@Filter` — аннотация указывает, что метод реализует функциональность фильтра сообщений. Обычно должно возвращаться булево значение.
- `@Transformer` — эта аннотация указывает, что метод реализует функциональность модификации сообщения, его заголовка и/или содержимого.
- `@ServiceActivator` — аннотация указывает, что метод способен обрабатывать сообщения.

Для запуска этого примера закомментируйте аннотацию `@ImportResource`. И все. В журнале должен появиться примерно следующий вывод:

```
...
INFO 43940 - [main] c.a.todo.TodoIntegrationApplication      : Started
TodoIntegrationApplication in 1.002 seconds (JVM running for 1.625)
INFO 43940 - [main] i.ToDoIntegration$SimpleServiceActivator : BUY MILK
TODAY
...
```

Использование JavaConfig

Использование JavaConfig очень напоминает проделанное выше. Для иллюстрации мы поменяем последнюю часть потока интеграции. Закомментируйте конечную точку сообщений внутреннего класса `SimpleServiceActivator` и замените ее следующим кодом:

```
@Bean
@ServiceActivator(inputChannel = "toLog")
public LoggingHandler logging() {
    LoggingHandler adapter = new
                  LoggingHandler(LoggingHandler.Level.INFO);
    adapter.setLoggerName("SIMPLE_LOGGER");
    adapter.setLogExpressionString
        ("headers.id + ': ' + payload");
    return adapter;
}
```

Этот код создает объект `LoggingHandler` — фактически тот же объект, что генерирует XML-конфигурация на основе тега `logging-channel-adapter`. Он заносит в журнал сообщение `SIMPLE_LOGGER` с идентификатором заголовка и содержимым, в данном случае с сообщением `BUY MILK TODAY`.

Опять же я понимаю, что это очень примитивный пример, но по крайней мере он дает вам представление о том, как работает Spring Integration и как его следует настраивать. Заказчики часто спрашивают меня, можно ли смешивать различные конфигурации. Разумеется! Мы сейчас это увидим.

Приложение ToDo с интеграцией чтения файлов

Посмотрим, как интегрировать чтение файлов. Это очень распространенная задача при интеграции систем, один из самых частых сценариев использования. Начнем с создания класса `ToDoProperties`, который помогает читать путь и имя файла во внешние свойства (листинг 11.7).

Листинг 11.7. com.apress.todo.config.ToDoProperties.java

```
package com.apress.todo.config;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

@Data
@ConfigurationProperties(prefix="todo")
public class ToDoProperties {

    private String directory;
    private String filePattern;
}
```

Как видите, ничего нового в листинге 11.7 нет. А поскольку наше приложение читает данные из файла, необходимо создать преобразователь, который бы читал запись типа String, производил ее синтаксический разбор и возвращал новый экземпляр `ToDo`. Создайте класс `ToDoConverter` (листинг 11.8).

Листинг 11.8. com.apress.todo.integration.ToDoConverter.java

```
package com.apress.todo.integration;
import com.apress.todo.domain.ToDo;
import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;

import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

@Component
public class ToDoConverter implements Converter<String, ToDo> {
    @Override
    public ToDo convert(String s) {
        List<String> fields = Stream.of(s.split(",")).map(String::trim).
        collect(Collectors.toList());
        return new ToDo(fields.get(0),Boolean.parseBoolean(fields.get(1)));
    }
}
```

Ничего особенного в листинге 11.8 нет. Единственное требование — он должен реализовывать обобщенный интерфейс `Converter`. Я поговорю об этом в следующем разделе. Еще один нужный нам класс — обработчик для экземпляра `ToDo`. Создайте класс `ToDoMessageHandler` (листинг 11.9).

Листинг 11.9. com.apress.todo.integration.ToDoMessageHandler.java

```
package com.apress.todo.integration;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

@Component
public class ToDoMessageHandler {
    private Logger log = LoggerFactory.getLogger(ToDoMessageHandler.class);

    public void process(ToDo todo){
        log.info(">>> {}", todo);
         // Дальнейшая обработка...
    }
}
```

Листинг 11.9 представляет собой простой класс POJO; метод, получающий на входе экземпляр `ToDo`.

Создадим основной поток интеграции. Создайте класс `ToDoFileIntegration` (листинг 11.10).

Листинг 11.10. com.apress.todo.integration.ToDoFileIntegration.java

```
package com.apress.todo.integration;

import com.apress.todo.config.ToDoProperties;
import org.springframework.boot.context.properties.
  EnableConfigurationProperties;
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.Pollers;
import org.springframework.integration.dsl.Transformers;
import org.springframework.integration.file.dsl.Files;
import org.springframework.integration.file.splitter.FileSplitter;

import java.io.File;
```

```
@EnableConfigurationProperties(ToDoProperties.class)
@Configuration
public class ToDoFileIntegration {

    private ToDoProperties props;
    private ToDoConverter converter;

    public ToDoFileIntegration(ToDoProperties props,
                               ToDoConverter converter) {
        this.props = props;
        this.converter = converter;
    }

    @Bean
         public IntegrationFlow fileFlow(){
              return IntegrationFlows
                .from(
             Files.inboundAdapter(
                         new File(this.props.getDirectory()))
                  .preventDuplicates(true)
                   .patternFilter(this.props.getFilePattern())
                         , e ->
                               e.poller(Pollers.fixedDelay(5000L))
                )
                .split(Files.splitter().markers())
                .filter(
                  p -> !(p instanceof FileSplitter.FileMarker))
                .transform(Transformers.converter(converter))

                .handle("toDoMessageHandler","process")
                .get();
    }
}
```

В листинге 11.10 показан основной поток интеграции, читающий содержимое файла (из файловой системы), преобразующий это содержимое в объект (в данном случае объект класса `ToDo` с помощью класса `ToDoConverter`) и обрабатывающий это сообщение на основе какой-то дополнительной логики. Проанализируем его подробнее.

- `from`. Перегруженный метод, в который обычно передается объект `MessageSource`; в данном случае мы передаем два значения: `Files.inboundAdapter` (о котором я расскажу позднее) и потребитель, получающий `SourcePollingChannelAdapterSpec`; в этом случае для опроса файловой системы каждые 5 секунд на предмет новых файлов с помощью класса `Pollers` используется лямбда-выражение.
- `Files`. Это готовый адаптер протокола; необходимо только задать его конфигурацию. Он используется для извлечения файлов из файловой системы.

Класс `Files` относится к Java DSL Spring Integration и предоставляет несколько удобных методов, в том числе `inboundAdapter`, обладающий текучим API, который возвращает объект класса `FileInboundChannelAdapterSpec`, в котором есть такие методы, как:

- `preventDuplicates` — позволяет задать аргумент равным `true` и избежать многократного чтения одного файла;
- `patternFilter` — выполняет поиск файлов, соответствующих паттерну.

В этом примере производится чтение из каталога (заданного значением свойства `todo.directory`) и из файла с соответствующим паттерну (заданного значением свойства `todo.file-pattern`) названием, оба — из файла `ToDoProperties`.

- `split`. Вызов этого метода указывает, что передаваемый ему параметр (которым может быть компонент, сервис, обработчик и т. п.) способен разбивать отдельное сообщение или содержимое сообщения на несколько сообщений или содержимых; в данном случае используется `FileMarker`, служащий разделителем при последовательной обработке файла.
- `filter`. Вследствие использования маркеров в качестве меток начала и конца сообщений мы получаем сперва маркер `FileMarker` начала, потом фактическое содержимое, а затем `FileMarker` окончания, так что здесь мы запрашиваем только содержимое, но не маркер.
- `transform`. Для модификации сообщения здесь используется класс `Transformers`, у которого есть несколько реализаций и преобразователь (пользовательский преобразователь, класс `ToDoConverter`, см. листинг 11.8).
- `handle`. Здесь мы используем класс, обрабатывающий сообщение с помощью передачи названия компонента (`toDoMessageHandler`) и отвечающий за выполнение процесса метода (взгляните на код класса `ToDoMessageHandler`, см. листинг 11.9). Класс `ToDoMessageHandler` представляет собой POJO, снабженный аннотацией `@Component`.

ПРИМЕЧАНИЕ

Java DSL Spring Integration поддерживает (на текущий момент) следующие классы адаптеров протоколов: amQp, JmS, Files, SFTP, FTP, HTTP, Kafka, Mail, Scripts и Feed. Эти классы располагаются в пакете org.springframework.integration.dsl.*.

Добавьте в файл `application.properties` следующее содержимое:

```
todo.directory=/tmp/todo
todo.file-pattern=list.txt
```

Конечно, можно добавить любой паттерн каталога и/или файла. Вместо `list.txt` здесь может быть все что угодно. Если взглянуть снова на класс `ToDoConverter`, вы увидите, что он ожидает только два значения: описание и булево значение. Так, файл `list.txt` выглядит следующим образом:

```
buy milk today, true
read a book, false
go to the movies, true
workout today, false
buy some bananas, false
```

Для запуска приложения закомментируйте весь код из класса `ToDoIntegration`. После запуска вы должны увидеть примерно следующее.

```
INFO 47953 - [           main] c.a.todo.TodoIntegrationApplication      :
Started TodoIntegrationApplication in 1.06 seconds (JVM running for 1.633)
INFO 47953 - [ask-scheduler-1] c.a.todo.integration.ToDoMessageHandler  :
>>> ToDo(id=3037a45b-285a-4631-9cfa-f89251e1a634, description=buy milk
today, created=2018-09-01T19:29:38.309, modified=2018-09-01T19:29:38.310,
completed=true)
INFO 47953 - [ask-scheduler-1] c.a.todo.integration.ToDoMessageHandler  :
>>> ToDo(id=7eb0ae30-294d-49d5-92e2-d05f88a7befd, description=read a
book, created=2018-09-01T19:29:38.320, modified=2018-09-01T19:29:38.320,
completed=false)
INFO 47953 - [ask-scheduler-1] c.a.todo.integration.ToDoMessageHandler  :
>>> ToDo(id=5380decb-5a6f-4463-b4b6-1567361c37a7, description=go to the
movies, created=2018-09-01T19:29:38.320, modified=2018-09-01T19:29:38.320,
completed=true)
INFO 47953 - [ask-scheduler-1] c.a.todo.integration.ToDoMessageHandler  :
>>> ToDo(id=ac34426f-83fc-40ae-b3a3-0a816689a99a, description=workout
today, created=2018-09-01T19:29:38.320, modified=2018-09-01T19:29:38.320,
completed=false)
INFO 47953 - [ask-scheduler-1] c.a.todo.integration.ToDoMessageHandler  :
>>> ToDo(id=4d44b9a8-92a1-41b8-947c-8c872142694c, description=buy some
bananas, created=2018-09-01T19:29:38.320, modified=2018-09-01T19:29:38.320,
completed=false)
```

Как вы видите, интегрировать обработку файлов, читать их содержимое и применять к данным любую бизнес-логику очень просто.

Помните, я говорил выше, что можно смешивать способы конфигурации Spring Integration? Итак, как же можно задействовать аннотацию для обработки

сообщения? Можно воспользоваться в качестве части конфигурации аннотацией `@ServiceActivator`:

```
@ServiceActivator(inputChannel="input")
public void process(ToDo message){

}
```

Для использования этого метода активатора сервиса необходимо изменить поток интеграции. Замените эту строку:

```
handle("toDoMessageHandler","process")
```

вот этой:

```
.channel("input")
```

Если теперь перезапустить пример, вы получите те же результаты. Заметили, что мы не описали входной канал? Spring Integration обнаружил, что нам нужен этот канал, и создал его «за кулисами».

Для рассказа обо всех замечательных возможностях Spring Integration потребовалась бы отдельная книга; конечно, здесь я могу лишь описать самые основы — привести краткое введение в возможности интеграции различных систем.

Spring Cloud Stream

Пока что вы посмотрели на все имеющиеся методы обмена сообщениями и узнали, что использование фреймворка Spring и Spring Boot упрощает для разработчиков и архитекторов создание чрезвычайно надежных программных решений для обмена сообщениями. В этом разделе мы пойдем дальше; мы заглянем в разработку нативных облачных приложений, своеобразное предисловие к следующей главе.

В следующем разделе мы поговорим о Spring Cloud Stream и о том, чем эта новая технология может помочь при разработке ориентированных на обработку сообщений микросервисных приложений.

Spring Cloud

Прежде чем обсуждать внутреннее устройство и использование Spring Cloud Stream, поговорим про включающее его семейство проектов, Spring Cloud.

Spring Cloud — набор утилит для разработчиков, позволяющий создавать приложения, использующие все распространенные паттерны распределенных систем: конфигурирование, обнаружение сервисов, автоматические выключатели (circuit breakers), интеллектуальную маршрутизацию, микропрокси, шину управления, глобальные блокировки, распределенные сеансы, межсервисные вызовы, распределенный обмен сообщениями и многое, многое другое.

Существует несколько основанных на Spring Cloud проектов, в числе которых Spring Cloud Config, Spring Cloud Netflix, Spring Cloud Bus, Spring Cloud for Cloud Foundry, Spring Cloud Cluster, Spring Cloud Stream и Spring Cloud Stream App Starters.

Если хотите начать работать с какой-либо из этих технологий, можете добавить следующие разделы и зависимости в свой файл `pom.xml` (если вы используете Maven).

- Добавьте тег `<dependencyManagement/>` с указанием GA-выпуска[1], например:

```
<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-dependencies</artifactId>
            <version>Finchley.SR1</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
```

- Добавьте технологии, которые хотите использовать, в тег `<dependencies/>`, например:

```
<dependencies>
 <dependency>
   <groupId>org.springframework.cloud</groupId>
   <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
 </dependency>
 <!-- Прочие технологии -->
</dependencies>
```

Если вы используете Gradle, можете добавить следующее содержимое в свой файл `build.gradle`:

```
ext {
      springCloudVersion = 'Finchley.SR1'
```

[1] См. https://ru.wikipedia.org/wiki/ Стадии_разработки_программного_обеспечения#General_availability_/_общедоступность.

```
}
dependencyManagement {
          imports {
         mavenBom "org.springframework.cloud:spring-cloud-dependencies:$
             {springCloudVersion}"
        }
}

dependencies {
      // ...
       compile ('org.springframework.cloud:spring-cloud-starter-stream-
           rabbit')
      // ...
}
```

Если вы внимательно изучите файл `pom.xml` аннотации Spring Cloud, то обнаружите, что теперь соглашение об именах имеет вид `spring-cloud-starter-<используемая технология>`. Обратите также внимание, что мы добавили тег управления зависимостями, благодаря которому можно работать с транзитивными зависимостями и управлять версиями библиотек.

Spring Cloud Stream

Пришло время поговорить о самом модуле Spring Cloud Stream. Почему я не описываю другие упомянутые технологии? Что такого особенного в Spring Cloud Stream? Spring Cloud Stream — облегченный и ориентированный на обмен сообщениями микросервисный фреймворк, в основе которого лежат Spring Integration и Spring Boot (обеспечивает продуманную среду выполнения для простой конфигурации). С его помощью можно легко создавать готовые к корпоративному использованию приложения — программные решения для обмена сообщениями и интеграции. Он предоставляет простую декларативную модель отправки и получения сообщений с помощью либо RabbitMQ, или Apache Kafka.

Мне кажется, что одна из важнейших особенностей Spring Cloud Stream — разделение обмена сообщениями между генераторами и потребителями за счет создания готовых привязок. Другими словами, для генерации и потребления сообщений не нужно добавлять в приложение никакого кода, специфичного для каждого брокера. Добавьте в приложение нужные зависимости привязок (я объясню, как это делать, позднее) — и Spring Cloud Stream сам позаботится обо всех аспектах взаимодействия при подключении и обмене сообщениями.

Итак, взглянем на основные компоненты Spring Cloud Stream и научимся использовать этот фреймворк.

Основы Spring Cloud Stream

Вот основные компоненты Spring Cloud Stream.

- *Модель приложения.* Модель приложения представляет собой ядро, не зависящее от промежуточного ПО в том смысле, что приложение использует для взаимодействия входной и выходной каналы к внешним брокерам (в качестве пути передачи сообщений) посредством реализаций привязок.
- *Абстракция адаптера привязки.* Spring Cloud Stream предоставляет реализации адаптеров привязки Kafka и RabbitMQ. Эта абстракция обеспечивает возможность соединения приложений Spring Cloud Stream с промежуточным ПО. Но откуда данная абстракция знает пункты назначения сообщений? Дело в том, что она может выбирать пункты назначения динамически, во время выполнения, на основе каналов. Обычно их необходимо указать в файле `application.properties` в свойствах `spring.cloud.stream.bindings.[input|ouput].destination`. Я расскажу об этом, когда мы будем изучать примеры.
- *Постоянная публикация/подписка.* Взаимодействие приложения производится посредством известной модели публикации/подписки. Платформа Kafka следует своей модели «тема/подписчик», а RabbitMQ создает точку обмена для темы и необходимые привязки для каждой из очередей. Эта модель снижает сложность генераторов и потребителей.
- *Группы потребителей.* Рано или поздно вы столкнетесь с необходимостью масштабирования потребителей. Оно производится путем введения понятия групп потребителей (аналогичного функциональности групп потребителей Kafka): объединения нескольких потребителей в группу для балансировки нагрузки, благодаря чему масштабирование сильно упрощается.
- *Поддержка секционирования.* Spring Cloud Stream поддерживает секции данных, благодаря которым несколько генераторов могут отправлять данные нескольким потребителям с гарантией обработки общих для них данных одним экземпляром потребителя. Это повышает производительность и согласованность данных.
- *API адаптеров привязки.* Spring Cloud Stream предоставляет API взаимодействия — Binder SPI (интерфейс поставщика сервиса, service provider interface), в котором можно расширять основное ядро функциональности путем модификации исходного кода, что упрощает реализацию конкретных адаптеров привязки, например JMS, WebSockets и т. д.

В этом разделе мы обсудим подробнее модель программирования и адаптеры привязки. Если хотите узнать больше об остальных понятиях, обратитесь к справочной документации Spring Cloud Stream. Задача, которую я перед собой ставлю: показать вам, как приступить к созданию событийно-управляемых микросервисов с помощью Spring Cloud Stream. На рис. 11.4 показано, что именно мы будем обсуждать.

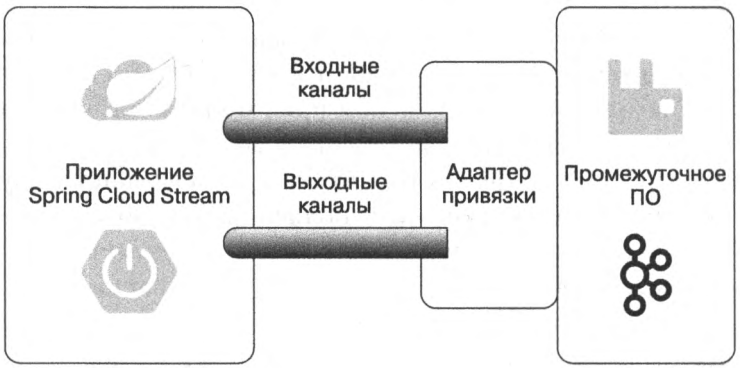


Рис. 11.4. Приложение Spring Cloud Stream

Программирование приложений Spring Cloud Stream

Что, исходя из рис. 11.4, нам нужно для создания приложения Spring Cloud Stream?

- *Тег* `<dependencyManagement/>`. Необходимо добавить в конфигурацию этот тег с последними версиями библиотек-зависимостей Spring Cloud.
- *Адаптер привязки*. Нужно выбрать подходящий тип адаптера привязки.
- *Kafka*. При использовании в качестве адаптера привязки Kafka необходимо добавить следующую зависимость в файл `pom.xml` (если вы используете Maven):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-kafka</artifactId>
</dependency>
```

Если вы используете Gradle, то нужно добавить следующую зависимость в файл `build.gradle`:

```
compile('org.springframework.cloud:spring-cloud-starter-stream-kafka')
```

- *RabbitMQ*. Если вы выбрали RabbitMQ в качестве адаптера привязки, необходимо добавить следующую зависимость в файл `pom.xml` (если вы используете Maven):

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

  Если вы используете Gradle, то нужно добавить следующую зависимость в файл `build.gradle`:

```
compile('org.springframework.cloud:spring-cloud-starter-stream-rabbit')
```

  Настройте и запустите Kafka или RabbitMQ. Можно ли использовать оба одновременно? Да, можно. Можете задать настройки для них в файле `application.properties`.

- `@EnableBinding`. Поскольку речь идет о приложении Spring Boot, то добавления аннотации `@EnableBinding` достаточно, чтобы превратить его в приложение Spring Cloud Stream.

В следующих разделах мы собираемся отправлять и получать сообщения из одного приложения в другое с помощью RabbitMQ в качестве транспортного слоя, без каких-либо знаний об API брокеров или настройке генераторов/потребителей сообщений.

В Spring Cloud Stream роль механизма отправки/получения сообщений играют каналы (входной/выходной). В приложении Spring Cloud Stream может быть произвольное число каналов, так что для обозначения потребителей и генераторов в Spring Cloud Stream есть две аннотации, `@Input` и `@Output`. Обычно класс `SubscribableChannel` снабжается аннотацией `@Input` для прослушивания входящих сообщений, а класс `MessageChannel` — аннотацией `@Output` для отправки сообщений.

Помните, я говорил вам, что Spring Cloud Stream основан на Spring Integration?

Чтобы не работать вручную с этими каналами и аннотациями, Spring Cloud Stream упрощает ситуацию с помощью трех интерфейсов, охватывающих большинство сценариев использования обмена сообщениями: `Source`, `Processor` и `Sink`. Эти интерфейсы обеспечивают («за кулисами») все необходимые приложению каналы (входные/выходные).

- `Source` (Источник). Интерфейс используется в приложениях, где данные потребляются из внешней системы (с помощью прослушивания в очереди, вызова REST, файловой системы, запроса к базе данных и т. д.)

и отправляются через выходной канал. Вот описание этого интерфейса из Spring Cloud Stream:

```
public interface Source {

  String OUTPUT = "output";

  @Output(Source.OUTPUT)
  MessageChannel output();
}
```

- `Processor` (Обработчик). Интерфейс можно использовать в приложениях, где необходимо начать прослушивать входной канал на предмет новых сообщений, обработать полученные сообщения (расширять их, модифицировать и т. д.) и отправить новые сообщения в выходной канал. Вот описание этого интерфейса из Spring Cloud Stream:

```
public interface Processor extends Source, Sink {

}
```

- `Sink` (Потребитель). Интерфейс подходит для приложений, где нужно начать прослушивать входной канал на предмет новых сообщений, произвести обработку и завершить поток данных (сохранить данные, запустить задание, занести вывод в консоль и т. д.). Вот описание этого интерфейса из Spring Cloud Stream:

```
public interface Sink {
  String INPUT = "input";
  @Input(Sink.INPUT)
  SubscribableChannel input();
}
```

- Модели, с которыми мы работаем, показаны на рис. 11.5 и 11.6.

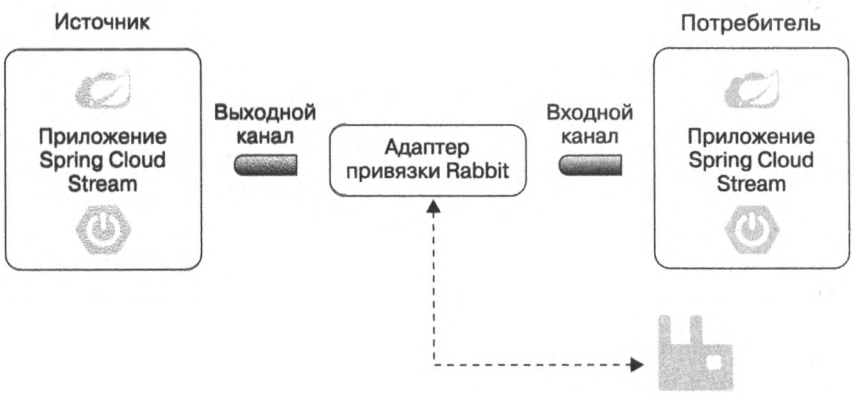


Рис. 11.5. Источник ▸ Потребитель

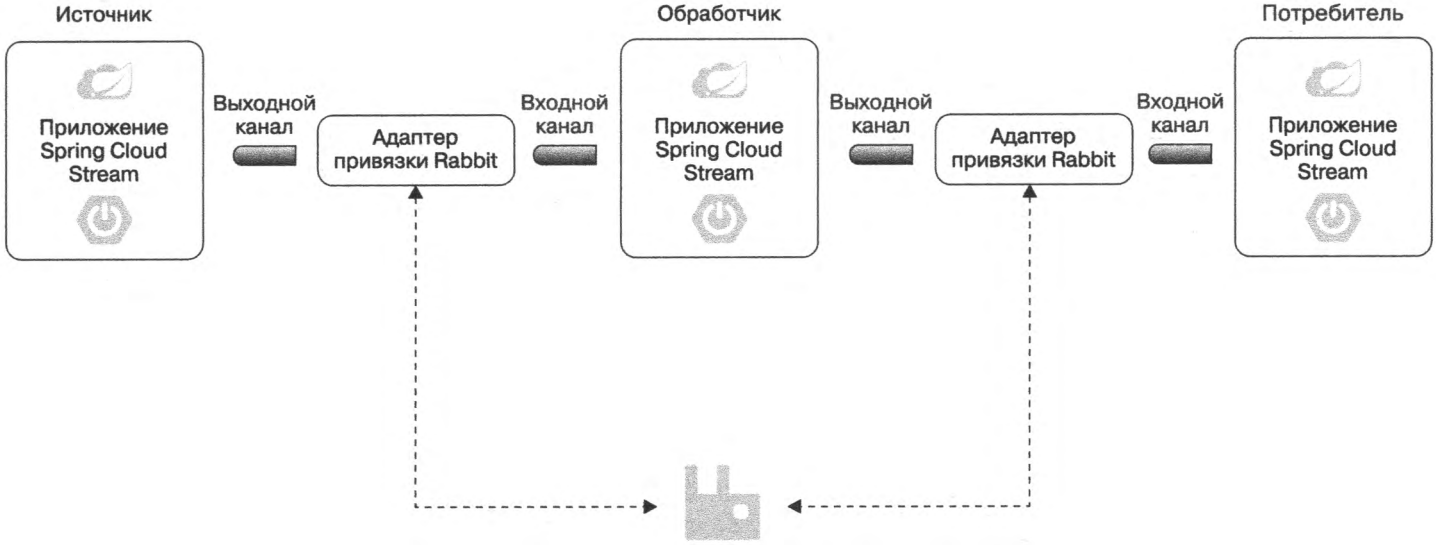


Рис. 11.6. Источник ▸ Обработчик ▸ Потребитель

Создание приложения ToDo с помощью Spring Cloud Stream

Цель этого проекта — продемонстрировать создание интерфейса `Source` и отправку сообщения через его выходной канал; создание интерфейса `Processor` и получение/отправку сообщения через его входной и выходной каналы соответственно; создания интерфейса `Sink` и получение сообщений из его входного канала. Я продемонстрирую описанное на рис. 11.6, прибавляя по одному приложению Spring Cloud Stream за раз.

Пока что все взаимодействие между этими приложениями происходит «в ручном режиме», в смысле промежуточных шагов, необходимых, чтобы вы могли разобраться, как работает каждое из этих приложений. В следующем разделе мы увидим работу потока данных в целом.

Можете начать приложение с нуля или просто следить за кодом в следующих разделах, чтобы узнать, что нужно делать. Если начинаете с нуля, то перейдите в Spring Initializr (https://start.spring.io) и внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.todo`.
- Artifact (Артефакт): `todo-cloud`.
- Name (Название): `todo-cloud`.
- Package Name (Название пакета): `com.apress.todo`.
- Dependencies (Зависимости): `Cloud Stream`, `Lombok`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами интегрированную среду разработки (рис. 11.7).

Можете воспользоваться классом `ToDo` из предыдущего проекта `todo-integration` (листинг 11.11).

Источник

Начнем с описания класса-источника. Напомню, что у этого компонента есть выходной канал. Создайте класс `ToDoSource`. Он должен выглядеть так, как показано в листинге 11.11.

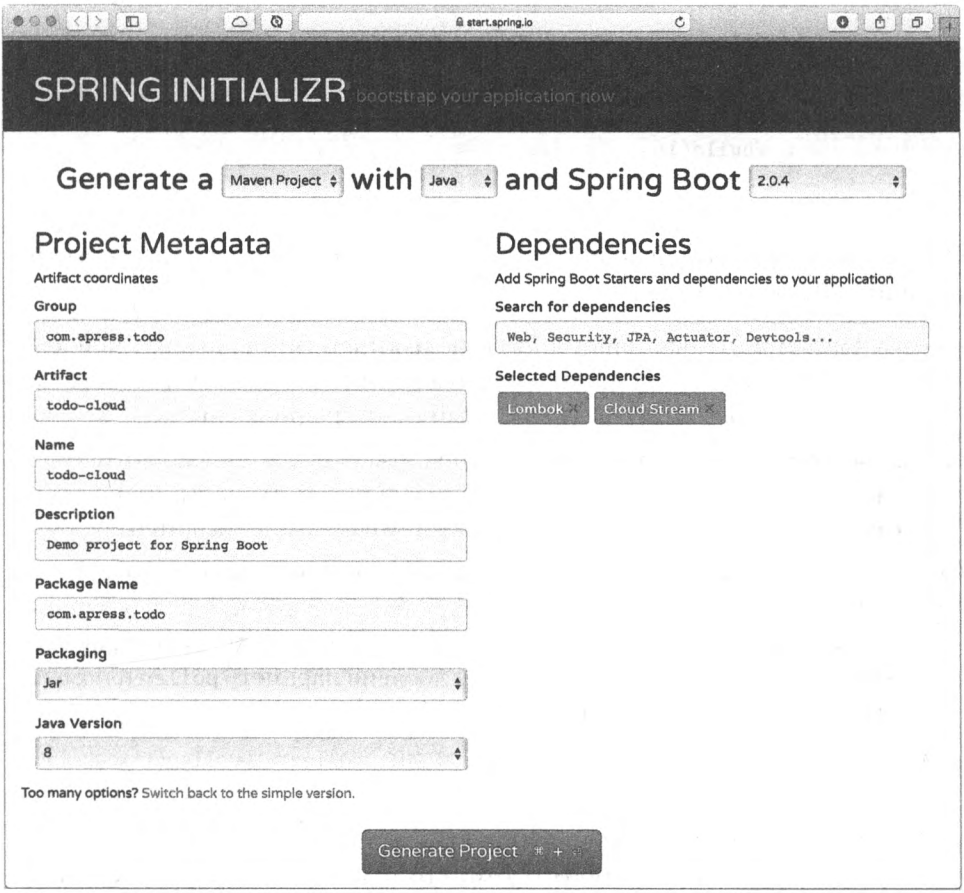


Рис. 11.7. Spring Initializr

Листинг 11.11. com.apress.todo.cloud.ToDoSource.java

```
package com.apress.todo.cloud;

import com.apress.todo.domain.ToDo;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.messaging.Source;
import org.springframework.context.annotation.Bean;
import org.springframework.integration.annotation.InboundChannelAdapter;
import org.springframework.integration.core.MessageSource;
import org.springframework.messaging.support.MessageBuilder;

@EnableBinding(Source.class)
public class ToDoSource {
```

```
    @Bean
    @InboundChannelAdapter(channel=Source.OUTPUT)
    public MessageSource<ToDo> simpleToDo(){
        return () -> MessageBuilder
                .withPayload(new ToDo("Test Spring Cloud Stream"))
                .build();
    }
}
```

В листинге 11.11 приведено простейшее возможное потоковое приложение-источник. Взглянем на него.

- `@EnableBinding`. Данная аннотация делает соответствующий класс приложением Spring Cloud Stream, активируя нужную конфигурацию для отправки/получения сообщений через указанный адаптер привязки.
- `Source`. Этот интерфейс указывает, что приложение Spring Cloud Stream является потоком данных — источником и создает нужные каналы; в данном случае выходной канал для отправки сообщений в указанный адаптер привязки.
- `@InboundChannelAdapter`. Эта аннотация входит в состав фреймворка Spring Integration. Она вызывает метод `simpleToDo` каждую секунду, в результате чего каждую секунду отправляется новое сообщение. Частоту отправки сообщений и их число можно поменять, добавив параметр `poller` и поменяв настройки по умолчанию; например:

  ```
  @InboundChannelAdapter(value = Source.OUTPUT, poller = @Poller(fixedDelay =
  "5000", maxMessagesPerPoll = "2"))
  ```

- Важную роль в этом объявлении играет канал; в данном случае значение `Source.OUTPUT` говорит о том, что используется выходной канал (`MessageChannel output()`).
- `MessageSource`. Интерфейс, служащий для отправки обратно `Message<T>` — обертки с содержимым и заголовками.
- `MessageBuilder`. Этот класс, служащий для отправки типа `MessageSource`, вам уже должен быть знаком; в данном случае отправляется сообщение с экземпляром `ToDo`.

Прежде чем запускать пример, учтите, что нужна зависимость для адаптера привязки, а поскольку мы хотим использовать RabbitMQ, то необходимо добавить в файл `pom.xml` (если вы применяете Maven) следующее:

```
<dependency>
  <groupId>org.springframework.cloud</groupId>
  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>
</dependency>
```

Если вы используете Gradle, добавьте следующую зависимость в файл `build.gradle`.

```
compile('org.springframework.cloud:spring-cloud-starter-stream-rabbit')
```

Убедитесь, что RabbitMQ запущен. И запустите пример. Вероятно, вы ничего особенного не увидите, но пример работает. И перейдите в RabbitMQ.

1. Откройте веб-консоль управления RabbitMQ в браузере. Перейдите по адресу http://localhost:15672. Имя пользователя — **guest**, пароль тоже **guest**.
2. Перейдите на вкладку Exchanges (Точки обмена) (рис. 11.8).

   Обратите внимание, что была создана точка обмена `output`, а частота обмена сообщениями — 1,0 в секунду.

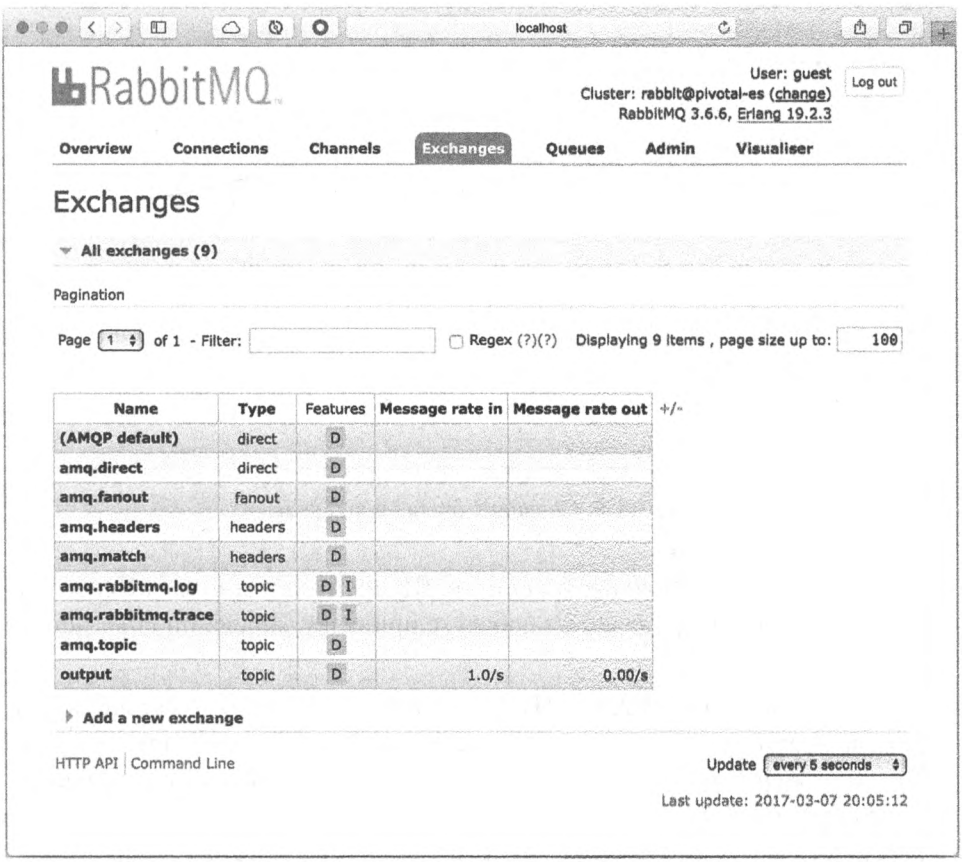


Рис. 11.8. Вкладка Exchanges (Точки обмена) RabbitMQ

3. Привяжем эту точку обмена к очереди; но сначала надо создать очередь. Перейдите на вкладку Queues (Очереди) и создайте новую очередь `my-queue` (рис. 11.9).

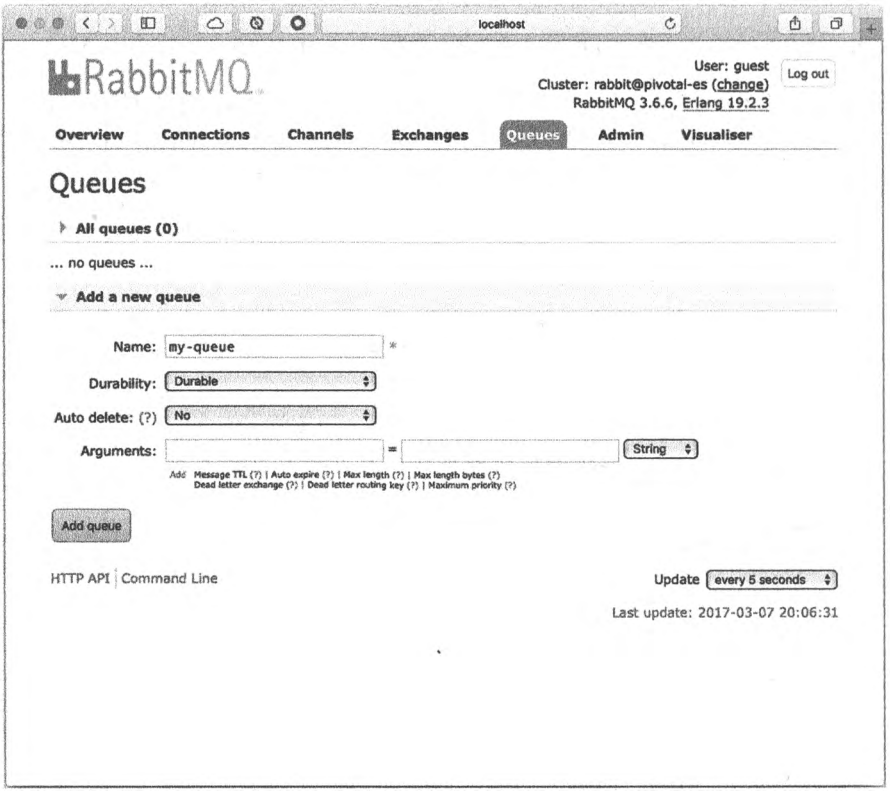


Рис. 11.9. Создание очереди my-queue

4. Созданная очередь появится в списке. Щелкните на `my-queue`. Перейдите в раздел Bindings (Привязки) и добавьте привязку. Значения показаны на рис. 11.10.
5. Заполните поле From Exchange (Из точки обмена) значением `output` (название выходной точки обмена). А поле Routing Key (Ключ маршрутизации) должно содержать значение `#`, в результате чего в `my-queue` смогут попасть все сообщения.
6. После привязки точки обмена `output` к очереди `my-queue` вы увидите несколько сообщений. Откройте панель Overview (Обзор) (рис. 11.11).

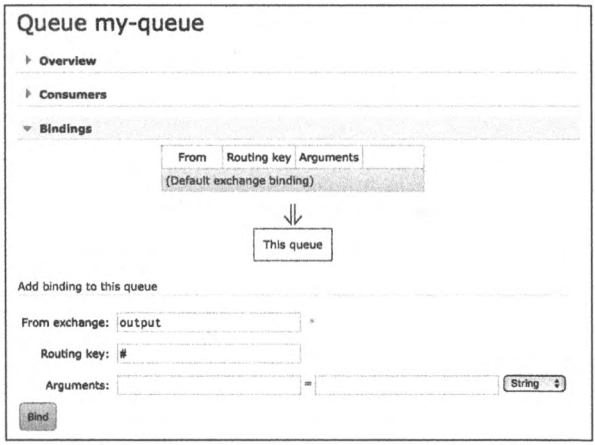


Рис. 11.10. Привязки

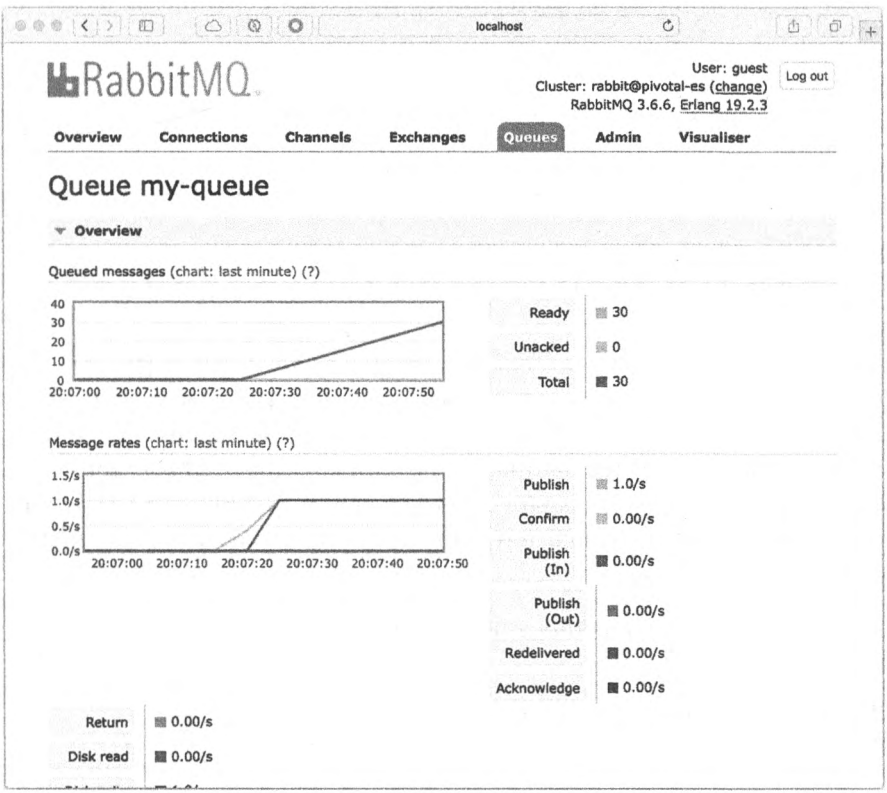


Рис. 11.11. Обзор

7. Взглянем на одно из сообщений. Откройте для этого панель Get Messages[1] (Получить сообщения). Вы можете получить любое количество сообщений и посмотреть их содержимое (рис. 11.12).

Рис. 11.12. Получение сообщений

Если вы получаете несколько сообщений, то взгляните на их разделы Payload (Содержимое). Мы получаем по сообщению каждую секунду. (Обратите внимание, что формат содержимого по умолчанию — JSON, а также что у сообщений есть свойства, например заголовки с `contentType: application/json` и `delivery_mode: 2`, означающим сохранение сообщения.) Именно так Spring Cloud Stream и его адаптер привязки подключаются к RabbitMQ для публикации сообщений.

Если взглянуть на сообщение, в нем будут видны даты со всеми подробностями:

```
{"id":"68d4100a-e706-4a51-a254-d88545ffe7ef","description":"Test Spring
Cloud Stream","created":{"year":2018,"month":"SEPTEMBER","hour":21,"minute
":9,"second":5,"nano":451000000,"monthValue":9,"dayOfMonth":2,"dayOfWeek":
```

[1] Для этого необходимо щелкнуть на my-queue на вкладке Queues (Очереди).

```
"SUNDAY","dayOfYear":245,"chronology":{"id":"ISO","calendarType":"iso8601
"}},"modified":{"year":2018,"month":"SEPTEMBER","hour":21,"minute":9,"seco
nd":5,"nano":452000000,"monthValue":9,"dayOfMonth":2,"dayOfWeek":"SUNDAY","
dayOfYear":245,"chronology":{"id":"ISO","calendarType":"iso8601"}},"complet
ed":false}
```

Здесь можно видеть очень подробную сериализацию дат, но такое поведение можно изменить, добавив в файл `pom.xml` (если вы используете Maven) следующую зависимость:

```
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jsr310</artifactId>
</dependency>
```

Если вы используете Gradle, добавьте в файл `build.gradle` следующую зависимость:

```
compile('com.fasterxml.jackson.datatype:jackson-datatype-jsr310')
```

Перезапустите приложение. Теперь вы увидите сообщения следующего вида:

```
{"id":"37be2854-91b7-4007-bf3a-d75c805d3a0a","description":"Test Spring
 Cloud Stream","created":"2018-09-02T21:12:12.415","modified":"2018-09-
 02T21:12:12.416","completed":false}
```

Обработчик

Для входного канала (куда поступают все новые сообщения) он использует `Listener`, получает сообщение `ToDo`, преобразует его описание в верхний регистр, отмечает запланированное дело как завершенное и затем отправляет его в выходной канал.

Создайте класс `ToDoProcessor`. Он должен выглядеть так, как показано на рис. 11.12.

Листинг 11.12. com.apress.todo.cloud.ToDoProcessor.java

```
package com.apress.todo.cloud;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.cloud.stream.messaging.Processor;
import org.springframework.messaging.handler.annotation.SendTo;
```

```
import java.time.LocalDateTime;

@EnableBinding(Processor.class)
public class ToDoProcessor {

    private Logger log = LoggerFactory.getLogger(ToDoProcessor.class);

    @StreamListener(Processor.INPUT)
        @SendTo(Processor.OUTPUT)
    public ToDo transformToUpperCase(ToDo message) {
        log.info("Processing >>> {}", message);
        ToDo result = message;
        result.setDescription(message.getDescription().toUpperCase());
        result.setCompleted(true);
        result.setModified(LocalDateTime.now());
        log.info("Message Processed >>> {}", result);
        return result;
    }
}
```

В листинге 11.12 приведен простой поток данных — обработчик. Проанализируем его.

- `@EnableBinding`. Эта аннотация делает соответствующий класс приложением Spring Cloud Stream, активируя нужную конфигурацию для отправки/получения сообщений через указанный адаптер привязки.
- `Processor`. Этот интерфейс указывает, что приложение Spring Cloud Stream является потоком данных — обработчиком. Он создает нужные каналы; в нашем случае входной канал (для прослушивания на предмет новых входящих сообщений) и выходные каналы (для отправки сообщений в указанный адаптер привязки).
- `@StreamListener`. Эта аннотация входит в состав фреймворка Spring Cloud Stream и очень похожа на аннотации `@RabbitListener` и `@JmsListener`. Она прослушивает на предмет входящих сообщений в канале `Processor.INPUT` (`SubscribableChannel input()`).
- `@SendTo`. Уже знакомая вам аннотация; она использовалась в предыдущей главе. Задача ее осталась прежней; можно считать, что она служит в качестве генератора для возврата ответа. Она отправляет сообщение в канал `Processor.OUTPUT` (`MessageChannel output()`).

Мне представляется, что это простейший, но достаточно хороший пример того, что можно делать с потоком данных `Processor`. Перед тем как его запустить,

не забудьте закомментировать аннотацию `@EnableBinding` в классе `ToDoSource` и удалить выходную точку обмена и очередь `my-queue`.

Запустите пример. Опять же приложение выполняет не так уж много действий, но перейдем в веб-консоль управления RabbitMQ.

1. Перейдите в браузер и откройте страницу http://localhost:15672 (имя пользователя — `guest`, пароль — `guest`).
2. Щелкните на вкладке Exchanges (Точки обмена), и вы увидите, что были созданы та же точка обмена `output` и новая точка входная обмена `input`. Как вы помните, поток данных `Processor` использует входной и выходной каналы (рис. 11.13).

   Обратите внимание, что ни в одной из новых точек обмена не указана частота сообщений.

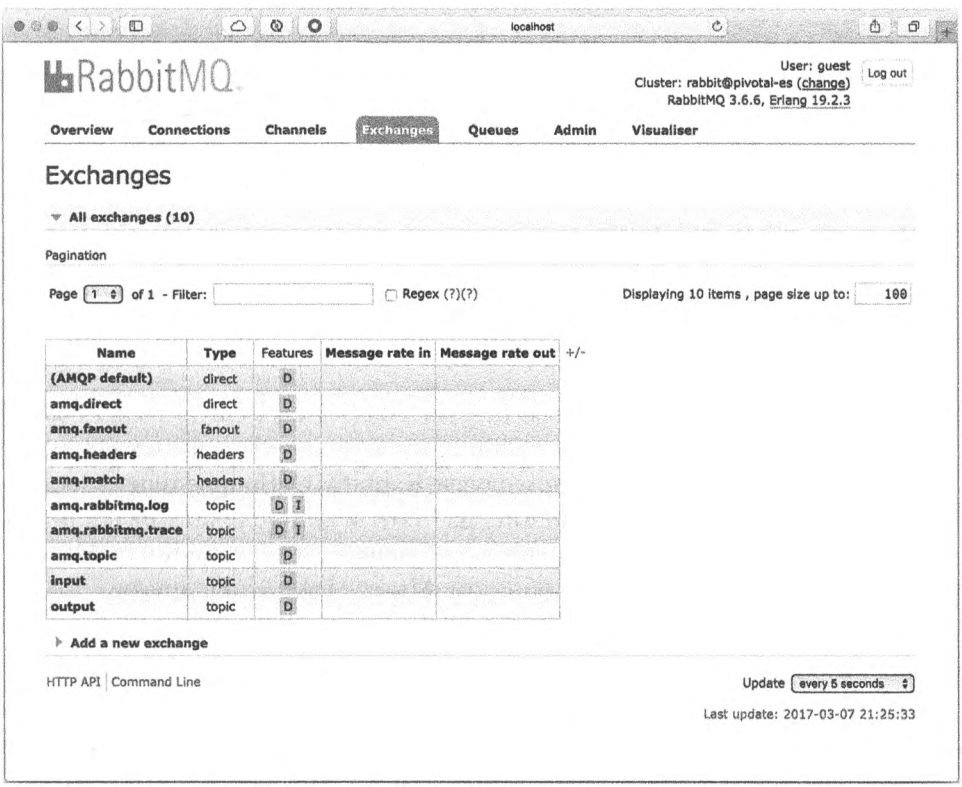


Рис. 11.13. Точки обмена

3. Перейдите на вкладку Queues (Очереди). Здесь вы должны увидеть новую очередь с названием `input.anonymous` плюс созданный случайный текст (рис. 11.14).

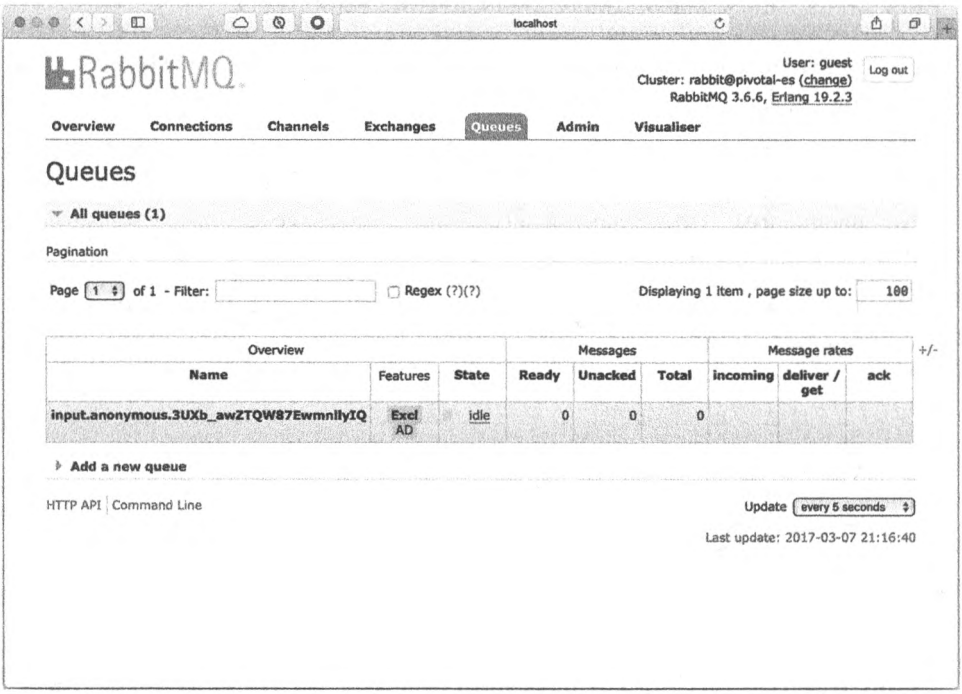


Рис. 11.14. Очереди

Вот и все. Поток данных `ToDoProcessor` создает выходную точку обмена и очередь `input.anonymous.*`, а значит, этот поток данных соединен с адаптером привязки, в данном случае RabbitMQ. Вопрос теперь состоит в том, как отправить сообщение, да? Это можно сделать различными способами: имитировать отправку сообщения с помощью RabbitMQ или отправить его программным образом. Мы сделаем и то и другое.

Мы создадим очередь `my-queue` и привяжем ее к точке обмена `output` аналогично тому, как мы делали в случае потока данных — источника.

4. Перейдите на вкладку Queues (Очереди), создайте очередь `my-queue` и привяжите ее к выходной точке обмена с ключом маршрутизации `#`. Все аналогично шагам 2 и 3 из потока данных `Source`. Отмечу также, что очередь `input.anonymous.*` привязана к точке обмена `input`.

5. А теперь мы отправим сообщение с помощью точки обмена `input`. Перейдите на вкладку Exchanges (Точки обмена). Щелкните на точке обмена input и выберите панель Publish Message (Опубликовать сообщение).

6. Введите в поле Payload (Содержимое) следующее:

```
{"id":"37be2854-91b7-4007-bf3a-d75c805d3a0a","description":
"Test Spring Cloud Stream","created":"2018-09-02T21:12:12.415",
"modified":"2018-09-02T21:12:12.416","completed":false}
```

   Введите в поле Properties (Свойства) `content-type=application/json` (рис. 11.15).

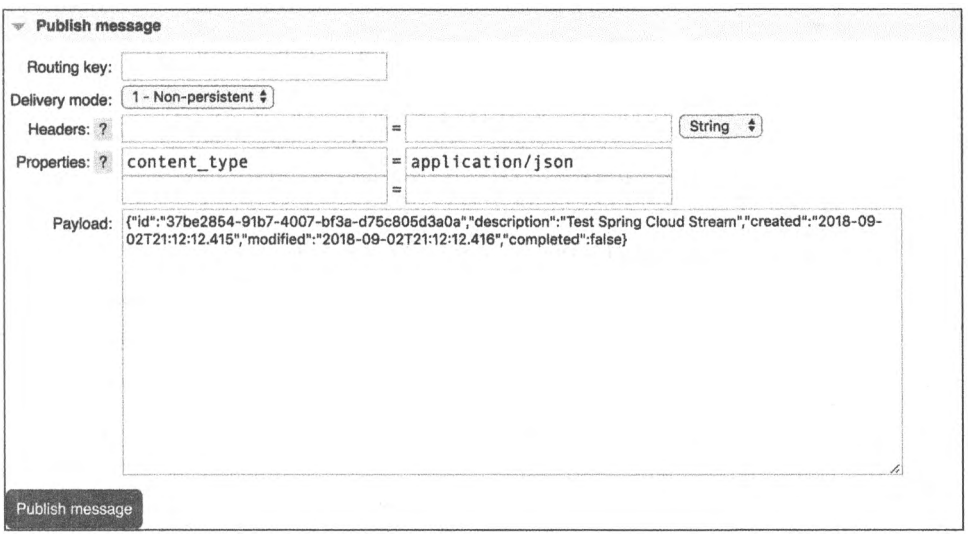


Рис. 11.15. Публикация сообщения

   Затем нажмите кнопку Publish Message (Опубликовать сообщение). Должно появиться всплывающее окно с надписью Message published (Сообщение опубликовано).

7. Взгляните в журнал приложения. Вы должны увидеть примерно следующее:

```
...
Processing >>> ToDo(id=37be2854-91b7-4007-bf3a-
d75c805d3a0a, description=Test Spring Cloud Stream,
created=2018-09-02T21:12:12.415, modified=2018-09-
02T21:12:12.416, completed=false)
Message Processed >>> ToDo(id=37be2854-91b7-4007-bf3a-
d75c805d3a0a, description=TEST SPRING CLOUD STREAM,
```

```
created=2018-09-02T21:12:12.415, modified=2018-09-
02T21:54:55.048, completed=true)
...
```

Если взглянуть на очередь `my-queue` и получить сообщение, вы увидите практически те же результаты (рис. 11.16).

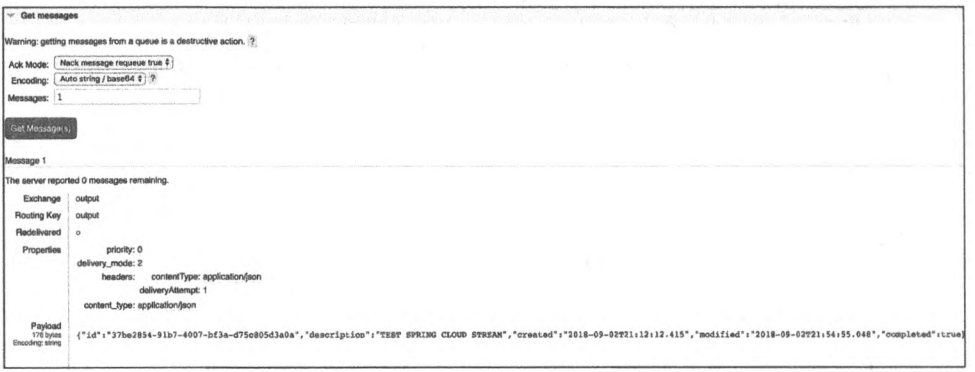


Рис. 11.16. Получение сообщения

Все просто, но неправильно. Вы никогда не будете отправлять сообщения с помощью консоли RabbitMQ, разве что в целях тестирования.

Я уже упоминал, что мы можем отправлять сообщения программным образом. Создайте класс `ToDoSender` (листинг 11.13).

Листинг 11.13. com.apress.todo.sender.ToDoSender.java

```
package com.apress.todo.sender;

import com.apress.todo.domain.ToDo;
import org.springframework.boot.ApplicationRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.support.MessageBuilder;

@Configuration
public class ToDoSender {

    @Bean
    public ApplicationRunner send(MessageChannel input){
        return args -> {
            input
```

```
                .send(MessageBuilder
                .withPayload(new ToDo("Read a Book"))
                .build());
        };
    }
}
```

Если запустить приложение, вы получите объект ToDo с описанием в верхнем регистре, описанный как завершенный в журнале и очереди `my-queue`. Как вы видите, мы используем знакомый вам по Spring Integration класс и интерфейс `MessageChannel`. Интересно, что Spring знает, какой канал внедрять. Напомню, что аннотация `@Processor` предоставляет входной канал.

Потребитель

Поток данных потребителя создает входной канал для прослушивания на предмет новых сообщений. Создадим класс `ToDoSink` (листинг 11.14).

Листинг 11.14. com.apress.todo.cloud.ToDoSink.java

```
package com.apress.todo.cloud;

import com.apress.todo.domain.ToDo;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cloud.stream.annotation.EnableBinding;
import org.springframework.cloud.stream.annotation.StreamListener;
import org.springframework.cloud.stream.messaging.Sink;

@EnableBinding(Sink.class)
public class ToDoSink {

    private Logger log = LoggerFactory.getLogger(ToDoSink.class);

    @StreamListener(Sink.INPUT)
    public void process(ToDo message){
        log.info("SINK - Message Received >>> {}",message);
    }
}
```

В листинге 11.14 приведен поток данных потребителя (Sink), его аннотации вам уже хорошо знакомы. Аннотация `@EnableBinding` преобразует этот класс в поток данных источника, который прослушивает новые входящие сообщения через канал `@StreamListener`, и канал `Sink.INPUT`. `Sink.INPUT` создает входной канал (`SubscribableChannel input()`).

Если воспользоваться листингом 11.13, закомментировать аннотацию `@EnableBinding` из класса `ToDoProcessor` и запустить приложение, то на панели управления RabbitMQ можно увидеть созданные точку обмена `input` и очередь `input.anonymous.*`, привязанные друг к другу. В журнале потребителя вы должны увидеть то же запланированное дело.

Помните, что поток данных потребителя производит дополнительные действия над полученным сообщением, но завершает поток данных.

Весь приведенный выше код не слишком функционален, он предназначен лишь для демонстрации концепции, что и было моей целью, поскольку я хотел показать вам, как это все работает «за кулисами». Теперь возьмем реальный сценарий использования, создадим полнофункциональный поток данных и посмотрим, как эти потоки данных взаимодействуют друг с другом без использования панели управления RabbitMQ.

Микросервисы

Я хотел бы рассказать о новом способе создания масштабируемых и высокодоступных приложений с помощью микросервисов. Важнейшее в этом разделе — возможность взаимодействия между потоками данных посредством обмена сообщениями. В итоге вы начнете рассматривать каждый поток данных (источник, обработчик и потребитель) как микросервис.

Приложение ToDo: полнофункциональный процесс

Перечислим некоторые из требований к этому новому приложению ToDo.

- Создание источника, который читает произвольные запланированные дела из файла, отфильтровывает завершенные и возвращает экземпляр ToDo.
- Создание узла-обработчика, принимающего на входе экземпляр ToDo и создающего текстовое сообщение.
- Создание потребителя, получающего на входе текст и отправляющего сообщение электронной почты получателю.

Как думаете, сможете это сделать?

На рис. 11.17 показан вполне реалистичный процесс (обратите внимание, что каждая его часть представляет собой отдельное приложение). Другими словами, необходимо создать приложения `todo-source`, `todo-processor` и `todo-sink`.

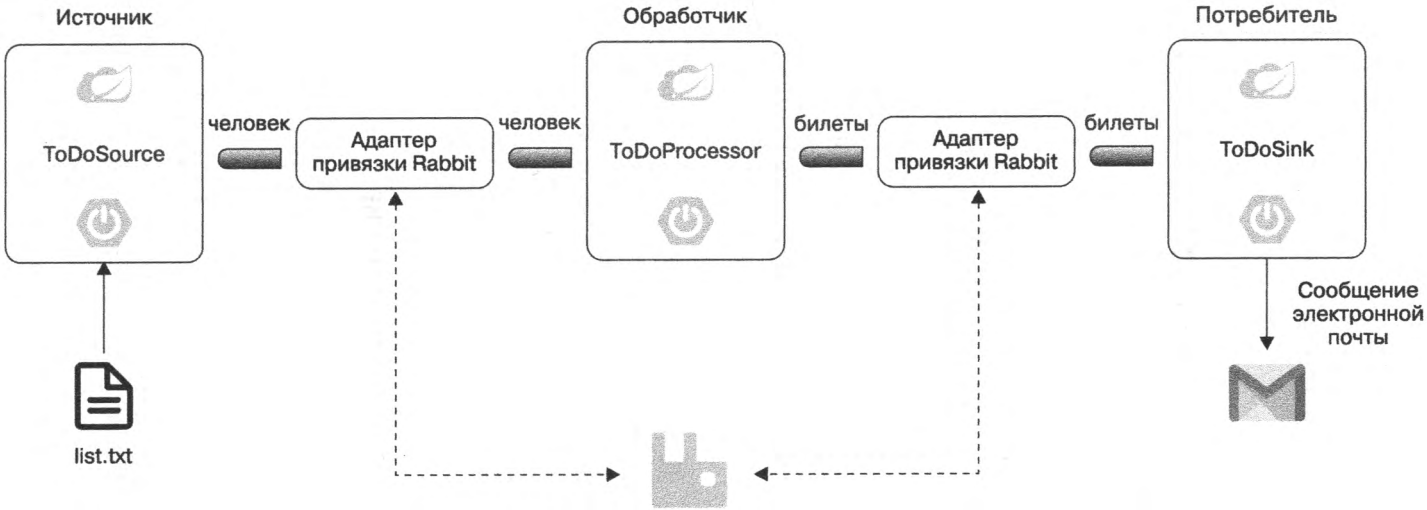


Рис. 11.17. Поток данных ToDo

Найдите в исходном коде главы 11 каждое из этих приложений. Сделайте так, чтобы они заработали, — это ваше домашнее задание. Поменяйте свойства конфигурации в соответствии с вашими настройками, в данном случае в проекте `todo-sink`.

«Стартовые пакеты» для приложений Spring Cloud Stream

Что, если я скажу вам, что можно было вместо создания предыдущего приложения просто воспользоваться так называемым стартовым пакетом (starter[1]) для приложений Spring Cloud Stream?

Spring Cloud Stream предоставляет готовые «стартовые пакеты» для приложений. Команда создателей Spring Cloud уже реализовала 52 приложения, которые можно скачать, настроить и запустить. Эти «стартовые пакеты» для приложений разбиты на три группы, по моделям `Source`, `Processor` и `Sink`.

- `Source`: file, ftp, gemfire, gemfire-cq, http, jdbc, jms, load-generator, loggregator, mail, mongodb, rabbit, s3, sftp, syslog, tcp, tcp-client, time, trigger, triggertask, twitterstream.
- `Processor`: bridge, filter, groovy-filter, groovy-transform, httpclient, pmml, scriptable-transform, splitter, tcp-client, transform и др.
- `Sink`: aggregate-counter, cassandra, counter, field-value-counter, file, ftp, gemfire, gpfdist, hdfs, hdfs-dataset, jdbc, log, rabbit, redis-pubsub, router, s3, sftp, task-launcher-local, task-launcher-yarn, tcp, throughput, websocket и многие другие.

ПРИМЕЧАНИЕ

Свежую версию «стартовых пакетов» для приложений можно найти в http://repo.spring.io/libs-release/org/springframework/cloud/stream/app/.

Справочную документацию по использованию других «стартовых пакетов» для приложений Spring Cloud Stream и их настройкам можно найти по адресу http://docs.spring.io/spring-cloud-stream-app-starters/docs/current/reference/html/.

[1] В русскоязычной литературе их также иногда называют «стартерами».

Резюме

В этой главе вы научились использовать Spring Integration и Spring Cloud Stream с помощью Spring Boot.

Вы узнали, как создавать с помощью Spring Integration надежные и хорошо масштабируемые приложения, которые можно интегрировать с другими системами.

Вы узнали, как благодаря Spring Cloud Stream можно легко создавать микросервисы. И научились использовать этот фреймворк с любым требуемым вам транспортным протоколом. Этот фреймворк не зависит от применяемого транспортного протокола и скрывает все нюансы обмена сообщениями; другими словами, для его использования вам не придется изучать RabbitMQ или Kafka.

В следующей главе я покажу, как Spring Boot может работать в облаке.

12 Spring Boot в облаке

Облачные вычисления — одно из важнейших понятий в IT-сфере. Компании, стремящиеся на передний край современных технологий, активно ищут способы повысить быстродействие своих сервисов. И стремятся к надежности в смысле как можно более быстрого восстановления от ошибок, так, чтобы клиенты ничего не заметили. Они стремятся к горизонтальному масштабированию (обычно под этим понимается наращивание вычислительных мощностей инфраструктуры, например добавление новых серверов, которые взяли бы на себя часть нагрузки) вместо вертикального (под которым понимается наращивание доступных ресурсов (CPU, оперативной памяти, дискового пространства и т. д.), существующих серверов). Какая же технология может обеспечить все это?

Решение этой задачи привело к появлению понятия *нативной облачной архитектуры* (cloud-native architecture), благодаря которой разработчики могут следовать паттернам, обеспечивающим быстродействие, надежность и масштабируемость. В этой главе я покажу вам, как создавать и развертывать приложения Spring Boot в облаке, следуя некоторым из этих паттернов.

Облачная и нативная облачная архитектура

Я думаю, вы слышали про эти компании: Pivotal, Amazon, Google, Heroku, Netflix и Uber. Они используют упомянутые мной выше идеи. Но как эти компании добиваются быстродействия, надежности и масштабируемости одновременно?

Одним из первопроходцев облачных вычислений была компания Amazon, которая начала рассматривать виртуализацию как основной инструмент для

обеспечения эластичности ресурсов; это означает возможность наращивания вычислительной мощности для любого развернутого приложения путем повышения числа виртуальных элементов, процессоров, объема оперативной памяти и т. д. без какого-либо участия IT-специалиста. Все эти новые способы масштабирования приложений понадобились, чтобы удовлетворить растущие требования пользователей.

А как удовлетворяет требования своих пользователей Netflix? Речь ведь идет о миллионах пользователей потокового мультимедийного контента ежедневно.

У всех этих компаний была готова необходимая для облачной эры инфраструктура, но не кажется ли вам, что любое приложение для работы в облаке также необходимо приспособить к этой новой технологии? Требуется учитывать, как масштабирование ресурсов влияет на приложение. Необходимо мыслить в терминах распределенных систем, правда? Думать о том, как приложения взаимодействуют с устаревшими системами и друг с другом в подобной среде. Что произойдет в случае сбоя одной из систем? Как восстановиться после сбоя? Как пользователи (а их миллионы) используют преимущества облачных вычислений?

На все эти вопросы дает ответ новая нативная облачная архитектура. Помните, что приложения должны быть быстрыми, надежными и масштабируемыми.

Во-первых, необходимо обеспечить прозрачность этой новой облачной среды в смысле улучшения мониторинга приложений — задания оповещений, инструментальных панелей и т. д. Требуется локализация сбоев и отказоустойчивость в смысле ограничения контекста приложений и отсутствия зависимостей между приложениями. В случае сбоя одного из приложений остальные должны продолжать работать. Непрерывное развертывание приложения не должно влиять на систему в целом. Это значит, что необходимо заранее позаботиться об автоматическом восстановлении, при котором система в целом способна обнаружить источник сбоя и произвести восстановление.

Приложения на основе 12 факторов

Инженеры компании Heroku выявили множество паттернов, ставших в итоге руководством по созданию приложений на основе 12 факторов (https://12factor.net/ru/). Это руководство демонстрирует, что приложение (отдельный модуль) должно основываться на декларативной конфигурации,

не сохранять состояние и не зависеть от способа развертывания. Приложение должно быть быстрым, надежным и масштабируемым.

Вот краткое резюме руководства по созданию приложений на основе 12 факторов.

- *Кодовая база.* Одной кодовой базе, отслеживаемой в системе контроля версий, может соответствовать много развертываний. У одного приложения должна быть единая кодовая база, отслеживаемая системой контроля версий (VCS), например Git, Subversion, Mercurial и т. д. На основе одной и той же кодовой базы можно производить много развертываний для среды разработки, тестирования, предэксплуатационного тестирования и промышленной эксплуатации.
- *Зависимости.* Объявляйте зависимости явным образом и изолируйте их. Иногда у среды отсутствует соединение с Интернетом (если речь идет о закрытой системе), так что необходимо заранее подумать о включении в пакет зависимостей (JAR-файлов, файлов Gem, совместно используемых библиотек и т. д.). При наличии внутреннего репозитория библиотек можно объявить манифест, например pom, gemfile, bundle и т. д. Никогда не полагайтесь на наличие чего-либо в итоговой среде.
- *Конфигурация.* Храните конфигурацию в среде выполнения. Не «зашивайте» в код ничего, что потенциально может меняться. Используйте переменные среды или сервер конфигурации.
- *Сторонние службы* (backing services). Считайте сторонние службы подключаемыми ресурсами. Подключайтесь к сервисам по URL или с помощью конфигурации.
- *Сборка, выпуск, запуск.* Четко разделяйте этапы сборки и запуска. Этот фактор связан с концепцией CI/CD (непрерывная интеграция, непрерывная доставка).
- *Процессы.* Приложение должно выполняться как один или более процессов без сохранения состояния. В процессах не должно сохраняться внутреннее состояние. Никакого разделения ресурсов. Любое нужное для работы приложения состояние должно рассматриваться как сторонний сервис.
- *Привязка портов.* Экспортируйте сервисы посредством привязки портов. Приложение должно быть автономным, а доступ к этим приложениям должен осуществляться через привязку портов. Приложение может стать сервисом другого приложения.

- *Параллелизм*. Масштабирование приложения должно производиться через модели процессов. Масштабируйте путем добавления дополнительных экземпляров приложения. Отдельные процессы могут быть многопоточными.
- *Утилизируемость*. Максимальная надежность благодаря быстрому запуску и корректному завершению. Процессы должны также быть утилизируемыми (помните: в них отсутствует сохранение состояния). Должна обеспечиваться отказоустойчивость.
- *Функциональная совместимость сред*. Среды разработки, предэксплуатационного тестирования и промышленной эксплуатации должны быть максимально похожи. Это обеспечивает непрерывную поставку.
- *Журналы*. Рассматривайте журналы как потоки событий. Приложение должно производить запись в *stdout*. Журналы — потоки агрегированных, упорядоченных по времени событий.
- *Процессы для администрирования*. Задачи, связанные с администрированием и управлением, должны запускаться в виде однократных процессов. Административные процессы должны выполняться на соответствующей платформе: миграция базы данных, одноразовые сценарии и т. д.

Микросервисы

Понятие *микросервисов* — новая парадигма разработки приложений. Микросервисы следует рассматривать как способ разбиения монолитных приложений на различные и независимые компоненты, следующие руководству по созданию приложений 12 факторов. Они работают после развертывания (рис. 12.1).

Мое мнение: микросервисы появились одновременно с изобретением Unix, поскольку утилиты командной строки, например grep, можно использовать как отдельное отлично работающее приложение. Причем можно создать лучшее приложение (систему) путем сочетания нескольких таких команд (например, `find . -name microservices.txt | grep -i spring-boot`). Но при этом данные команды независимы друг от друга и взаимодействуют с помощью конвейеров Unix (|). Внутри ваших приложений все аналогично.

Микросервисы ускоряют разработку. Почему? Поскольку можно выделить маленькую группу людей для разработки одной функциональной

возможности приложения с ограниченным контекстом, следующей руководству по созданию приложений 12 факторов.

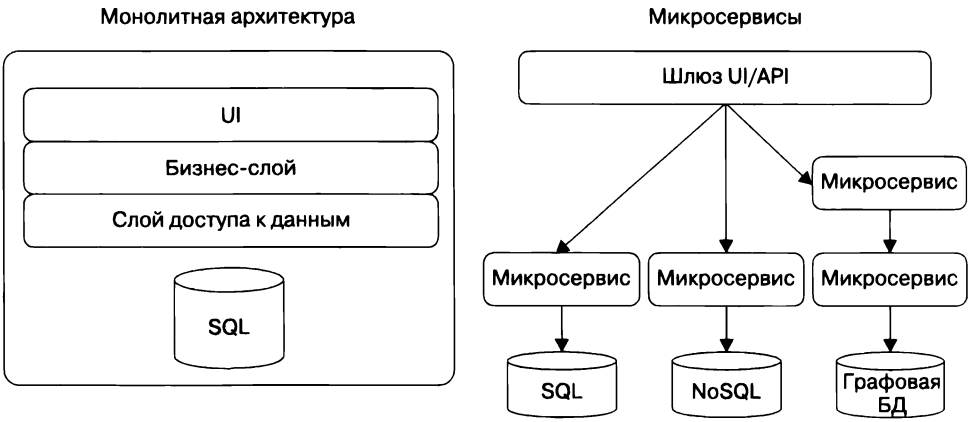


Рис. 12.1. Монолитная архитектура и микросервисы

Можно немало всего рассказать о микросервисах и миграции существующих архитектур на микросервисы, но наша задача — изучить Spring Boot и научиться развертывать приложения Spring Boot в облачной среде.

Подготовка приложения ToDo как микросервиса

Что нужно для превращения приложения ToDo Spring Boot в микросервис? На самом деле ничего. Да, вы не ослышались, ничего, поскольку Spring Boot позволяет с легкостью создавать микросервисы. Так что мы развернем то же самое приложение ToDo на облачной платформе. Какой именно платформе? Cloud Foundry компании Pivotal.

Можете взять проект `todo-rest` из предыдущих глав. Просмотрите его снова, если вы меняли что-то, и убедитесь, что он работает.

Важно убедиться в наличии следующих зависимостей; если вы используете Maven, в вашем файле `pom.xml` должны присутствовать следующие зависимости:

```
...
<dependency>
      <groupId>com.h2database</groupId>
      <artifactId>h2</artifactId>
      <scope>runtime</scope>
</dependency>
<dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <scope>runtime</scope>
</dependency>
...
```

Если вы используете Gradle, взгляните, есть ли в вашем файле `build.gradle` следующие зависимости:

```
...
runtime('com.h2database:h2')
runtime('mysql:mysql-connector-java')
...
```

Почему эти зависимости так важны? Вы узнаете об этом в следующих разделах. Далее перейдите в файл `application.properties` и убедитесь, что он выглядит таким образом:

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Тут изменилось значение свойства `ddl-auto`; раньше мы использовали значение `create-drop`, в результате чего схема создавалась и уничтожалась в конце сеанса. А теперь мы поменяли значение этого свойства на `update`, означающее обновление схемы по мере необходимости. Это не случайно, как мы увидим на деле в следующих разделах.

Подготовим приложение, выполнив следующую команду в каталоге с исходным кодом (можете также выполнить цель Maven или задачу Gradle в IDE; как это сделать, можно найти в документации). Если вы используете Maven, можете выполнить:

```
$ ./mvnw clean package
```

Если вы применяете Gradle, можете выполнить:

```
$ ./gradlew clean build
```

Эти команды создают JAR-файл, который мы очень скоро развернем. Так что придержите его пока что; мы скоро к нему вернемся.

ПРИМЕЧАНИЕ

Если у вас возникли проблемы с Java-конструктором для класса предметной области ToDo, значит, вы используете старую версию Lombok (поскольку класс предметной области снабжен аннотацией @NoArgsConstructor). Команда разработчиков Spring Boot пока не обновила эту библиотеку, так что используйте Lombok версии 1.18.2 или более поздней.

Платформа Pivotal Cloud Foundry

Платформа Cloud Foundry появилась в 2008 году и началась в качестве проекта компании VMWare с открытым исходным кодом, а затем перешла к Pivotal в 2013 году[1]. С тех пор Cloud Foundry является наиболее широко используемой PaaS. У Cloud Foundry, как решения с открытым исходным кодом, наибольшая поддержка со стороны сообщества разработчиков. Ее поддерживает несколько IT-компаний, включая IBM (с их платформой Bluemix), Microsoft, Intel, SAP и, конечно, Pivotal (с Pivotal Cloud Foundry – PAS и PKS), а также VMware.

Cloud Foundry – единственное решение с открытым исходным кодом, которое можно скачать и запустить без каких-либо проблем. Существует две версии Cloud Foundry: с открытым исходным кодом на сайте www.cloudfoundry.org и PAS и PKS от Pivotal (коммерческая версия) на сайте http://pivotal.io/platform. Причем если вас заинтересует коммерческая версия, то ее можно скачать с https://network.pivotal.io/products/pivotal-cf без каких-либо пробных версий или ограниченного демонстрационного периода. Фактически это тоже бесплатная версия, но если вам нужна поддержка или помощь в установке, то придется обратиться к менеджеру по продажам Pivotal.

В начале 2018 года Pivotal выпустила версию 2.0 платформы с расширенными возможностями для конечного пользователя. В ней на рынок были выведены решения Pivotal Application Service (сервис приложений Pivotal, PAS) и Pivotal Container Service (сервис контейнеров Pivotal, PKS, в основе которого лежит Kubernetes) (рис. 12.2).

[1] А в конце 2019 года снова вернулась в состав VMWare.

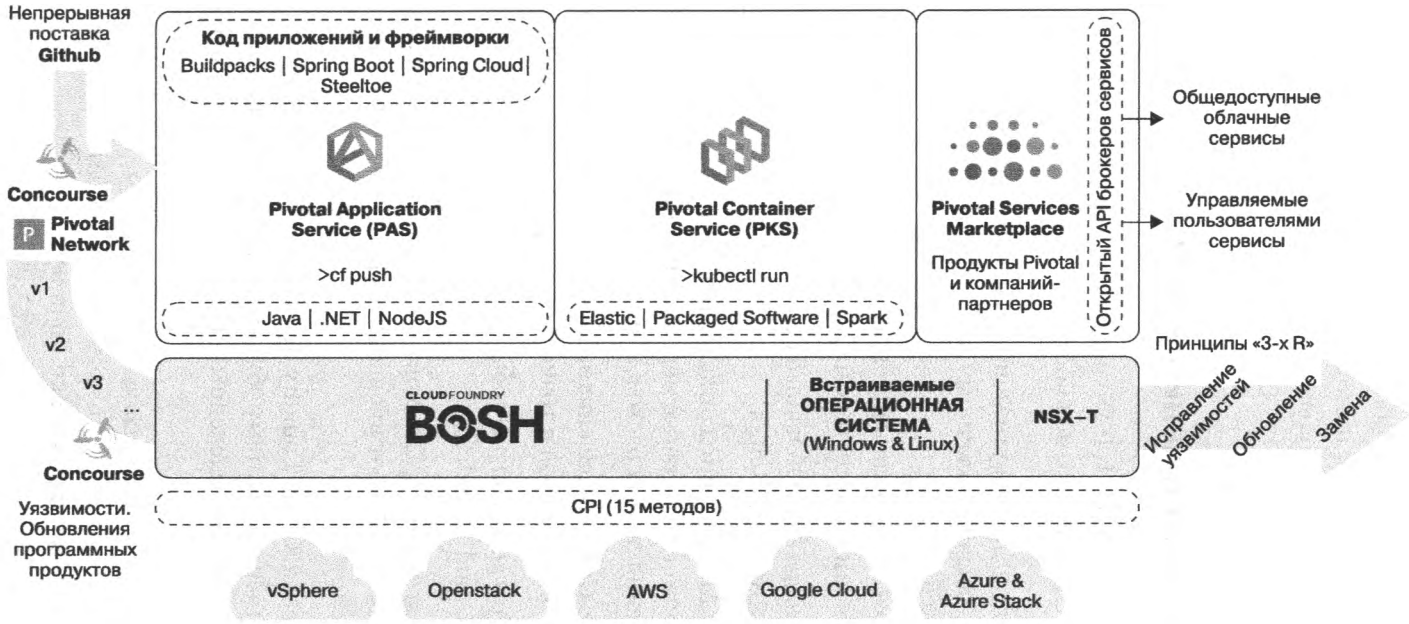


Рис. 12.2. Pivotal Cloud Foundry 2.0

В следующих разделах будет обсуждаться только PAS и будет описано, как можно просто приступить к нативной облачной разработке, поскольку вам нужно будет позаботиться только о своем приложении, о данных и больше ни о чем!

PAS: сервис приложений Pivotal

Сервис приложений Pivotal (PAS) построен на основе открытой архитектуры и предлагает следующие возможности.

- *Маршрутизатор.* Маршрутизация входящего трафика к соответствующему компоненту, обычно облачному контроллеру или запущенному на узле DEA приложению.
- *Аутентификация.* Управление идентификационной информацией обеспечивается совместно сервером OAuth2 и сервером входа в систему.
- *Облачный контроллер.* Облачный контроллер отвечает за управление жизненным циклом приложения.
- *Мониторинг.* Мониторинг, согласование состояний, версий и числа экземпляров приложений, а также перенаправление на облачный контроллер для исправления любых расхождений.
- *Garden/Diego Cells.* Управление экземплярами приложений, отслеживание запущенных экземпляров и трансляция сообщений о состоянии.
- *Хранилище больших двоичных объектов.* Ресурсы, код приложения, сборки и дроплеты Cloud Foundry.
- *Брокеры сервисов.* Брокеры сервисов отвечают за предоставление экземпляра сервиса, когда разработчик вводит сервис в эксплуатацию и связывает его с приложением.
- *Шина сообщений.* Cloud Foundry использует NATS (отличный от сетевого NATS), облегченную систему обмена сообщениями посредством публикации/подписки и распределенных очередей для внутренней связи между компонентами.
- *Журналирование и статистика.* Коллектор метрик собирает метрики компонентов. Пользователи могут задействовать эту информацию для мониторинга экземпляров Cloud Foundry.

Возможности PAS

Основанный на Cloud Foundry (с открытым исходным кодом) PAS обеспечивает готовые возможности PaaS на нескольких инфраструктурах с ведущим приложением и сервисами данных, в числе которых:

- коммерческая версия на основе Cloud Foundry с открытым исходным кодом;
- полная автоматизация развертывания, обновлений и осуществляемого одним нажатием мыши горизонтального и вертикального масштабирования на vSphere, vCloud Air, AWS, Microsoft Azure, Google Cloud и OpenStack с минимальным временем простоя в промышленной эксплуатации;
- мгновенное горизонтальное масштабирование прикладного слоя;
- веб-консоль для управления ресурсами и администрирования приложений и сервисов;
- преимущества, извлекаемые приложениями из встроенных сервисов, например балансировка нагрузки и DNS, автоматический контроль состояния приложения, журналирование и аудит;
- поддержка Java Spring посредством предоставляемого buildpack для запуска приложений Java;
- оптимизация работы разработчика с фреймворком Spring;
- сервис MySQL для быстрой разработки и тестирования;
- автоматическая привязка приложений и ввод в действие сервисов Pivotal, например Pivotal RabbitMQ, Pivotal Redis, Pivotal Cloud Cache (основан на GemFire) и MySQL для Pivotal Cloud Foundry.

В чем различие между версией с открытым исходным кодом и коммерческой? Фактически все перечисленные возможности. В версии с открытым исходным кодом необходимо все делать вручную, в основном с помощью командной строки (для установки, конфигурации, обновления и т. д.), а в коммерческой версии можно управлять своей инфраструктурой и запускать приложения с помощью веб-консоли. Знайте, что можно установить Cloud Foundry на Amazon AWS, OpenStack, Google Cloud, Microsoft Azure и vSphere. Pivotal Cloud Foundry (PAS и PKS) не зависит от используемой IaaS!

Использование PWS/PAS

Для использования PWS/PAS вам понадобится создать учетную запись в Pivotal Web Services по адресу https://run.pivotal.io/. Там можно получить пробную учетную запись[1] (рис. 12.3).

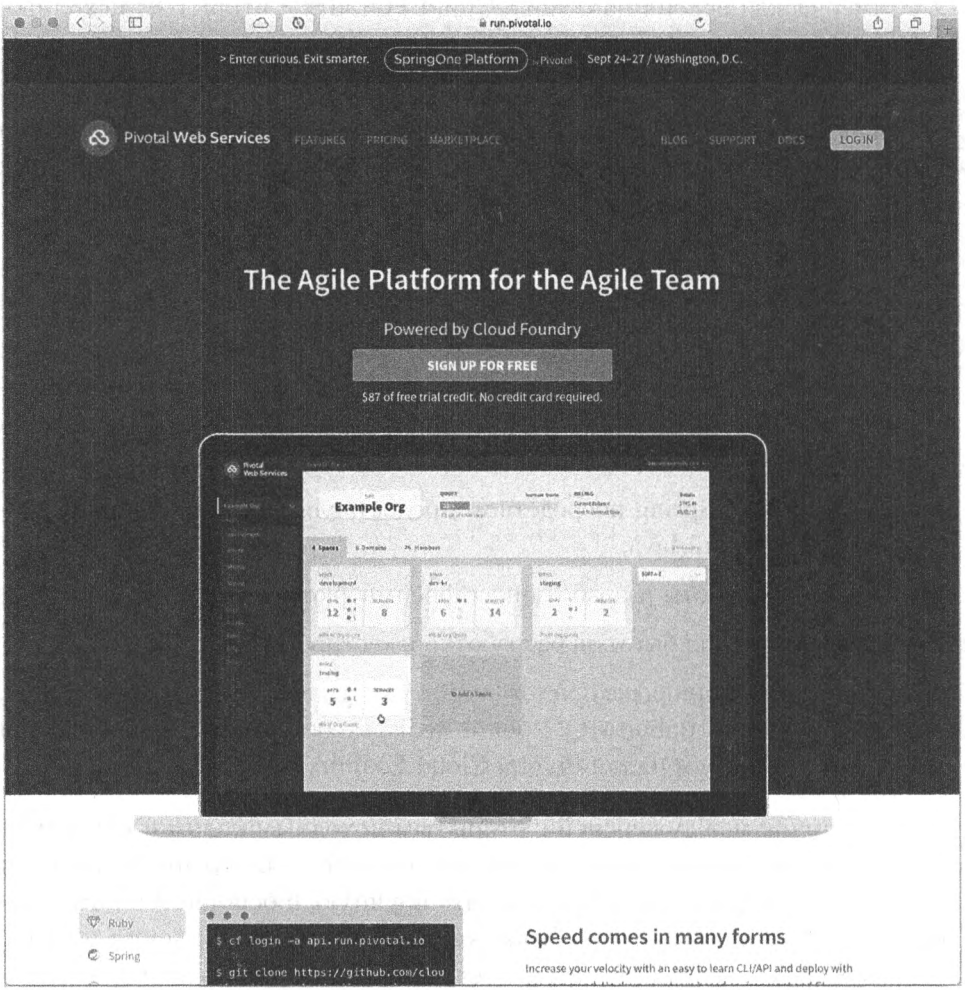


Рис. 12.3. Pivotal Web Services: https://run.pivotal.io/

[1] Это бесплатно и не требует указания номера кредитной карты, достаточно адреса электронной почты.

После регистрации вам нужно будет указать номер телефона, на который придет код активации вашей пробной учетной записи. Необходимо также указать организацию, можете ввести, например, свои инициалы с `-org`; например, я указал `fg-org`. По умолчанию после этого будет создано пространство (под названием `development`), в котором вы сможете работать (рис. 12.4).

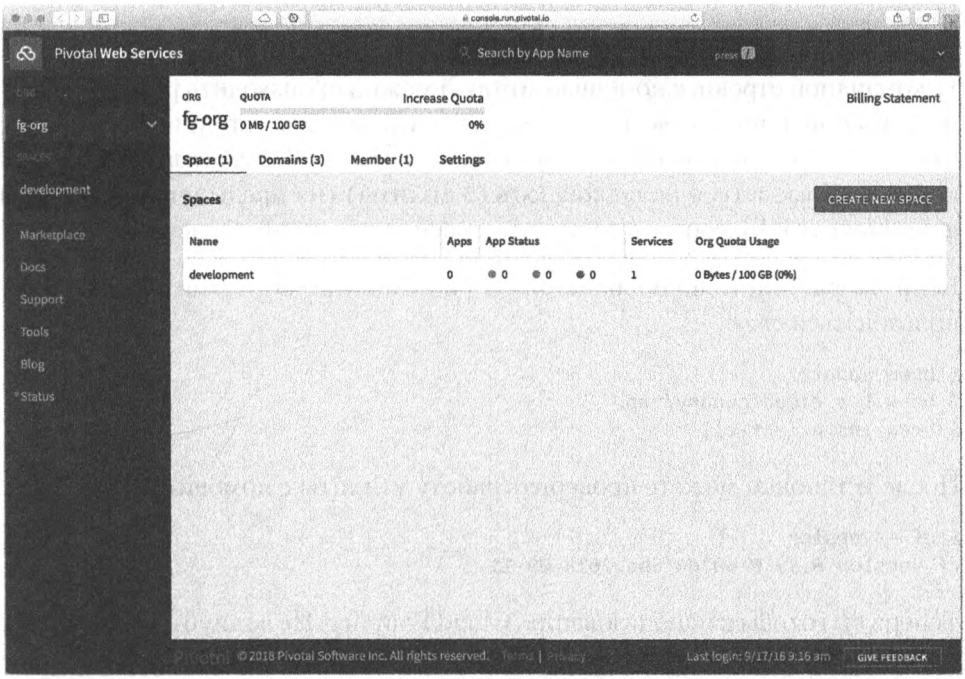


Рис. 12.4. Pivotal Web Services

Теперь можно развертывать приложения. По умолчанию, поскольку учетная запись пробная, у вас будет только 2 Гбайт оперативной памяти, но для развертывания приложения ToDo этого вполне достаточно. Обратите внимание на вкладки слева[1].

На вкладке Tools (Утилиты) можно найти ссылки для скачивания утилиты CLI (которую мы установим в следующем разделе), а также для входа в экземпляр PWS.

[1] Текущая версия сайта несколько отличается от описываемой здесь. В частности, в ней отсутствует вкладка Tools (Утилиты).

ПРИМЕЧАНИЕ

В следующем подразделе я буду попеременно использовать название PWS/PAS, подразумевая Cloud Foundry.

Cloud Foundry CLI: интерфейс командной строки

Прежде чем приступить к использованию PAS, необходимо установить утилиту командной строки, с помощью которой можно производить развертывание приложений и множество других задач. Если вы используете Windows, можете скачать ее свежую версию по адресу https://github.com/cloudfoundry/cli#downloads или воспользоваться вкладкой Tools (Утилиты) (из предыдущего раздела) и установить CLI, соответствующую вашей операционной системе.

Если вы работаете на операционной системе Mac OS/Linux, можете воспользоваться `brew`:

```
$ brew update
$ brew tap cloudfoundry/tap
$ brew install cf-cli
```

После установки можете проверить работу утилиты с помощью команды:

```
$ cf --version
cf version 6.39.0+607d4f8be.2018-09-11
```

Теперь вы готовы к использованию Cloud Foundry. Не волнуйтесь. Я покажу вам все основные команды, необходимые для развертывания и запуска приложения ToDo.

Вход в PWS/PAS с помощью утилиты CLI

Для входа в PWS и в свою учетную запись вам необходимо выполнить следующую команду:

```
$ cf login -a api.run.pivotal.io
API endpoint: api.run.pivotal.io

Email> your-email@example.org

Password>
Authenticating...
OK
```

```
Targeted org your-org
Targeted space development

API endpoint:   https://api.run.pivotal.io (API version: 2.121.0)
User:           your-email@example.org
Org:            your-org
Space:          development
```

По умолчанию вы попадаете в пространство `development`. Теперь вы готовы выполнять команды к PWS (экземпляру PAS) для создания, развертывания, масштабирования и т. д.

Развертывание приложения ToDo в PAS

Пора развернуть приложение ToDo в PAS. Учтите, что поддомен при развертывании должен быть уникальным. Мы обсудим это позднее.

Найдите JAR-файл приложения (`todo-rest-0.0.1-SNAPSHOT.jar`). Если вы используете Maven, он находится в каталоге `target`. Если вы применяете Gradle, он находится в каталоге `build/libs`.

Для отправки приложения используется следующая команда:

```
$ cf push <name-of-the-app> [options]
```

Так, для развертывания приложения ToDo можно выполнить следующее:

```
$ cf push todo-app -p todo-rest-0.0.1-SNAPSHOT.jar -n todo-app-fg

Pushing app todo-app to org your-org / space development as your-email@
example.org...
Getting app info...
Creating app with these attributes...
+ name:       todo-app
   path:        /Users/Books/pro-spring-boot-2nd/ch12/todo-rest/target/todo-
rest- 0.0.1-SNAPSHOT.jar
  routes:
+   todo-app-fg.cfapps.io
Creating app todo-app...
Mapping routes...
Comparing local files to remote cache...
Packaging files to upload...
...
...
     state     since       cpu     memory          disk
#0   running   T01:25:10Z  33.0%   292.7M of 1G    158.3M of 1G
```

У команды `cf` есть несколько опций.

- `-p` — указывает команде `cf` на необходимость загрузки на сервер файла или всего содержимого заданного каталога.
- `-n` — создает поддомен (должен быть уникальным). По умолчанию URI (должен быть уникальным) каждого приложения имеет вид `<поддомен>.cfapps.io`. Можно опустить опцию `-n`, но в команде `cf` указывается название приложения, которое может совпасть с другими названиями. В этом примере я использую название `todo-app-[мои_инициалы]` (`todo-app-fg`) и рекомендую вам поступить аналогичным образом.

«За кулисами» происходит вот что: приложение ToDo запускается в контейнере (не контейнере Docker), создаваемом RunC (https://github.com/opencontainers/runc), который использует ресурсы сервера и изолирован в целях безопасности. Теперь можете перейти в браузер и ввести заданный вами URI; в этом примере — https://todo-app-fg.cfapps.io/toDos.

Взглянем на наше приложение в PWS (рис. 12.5).

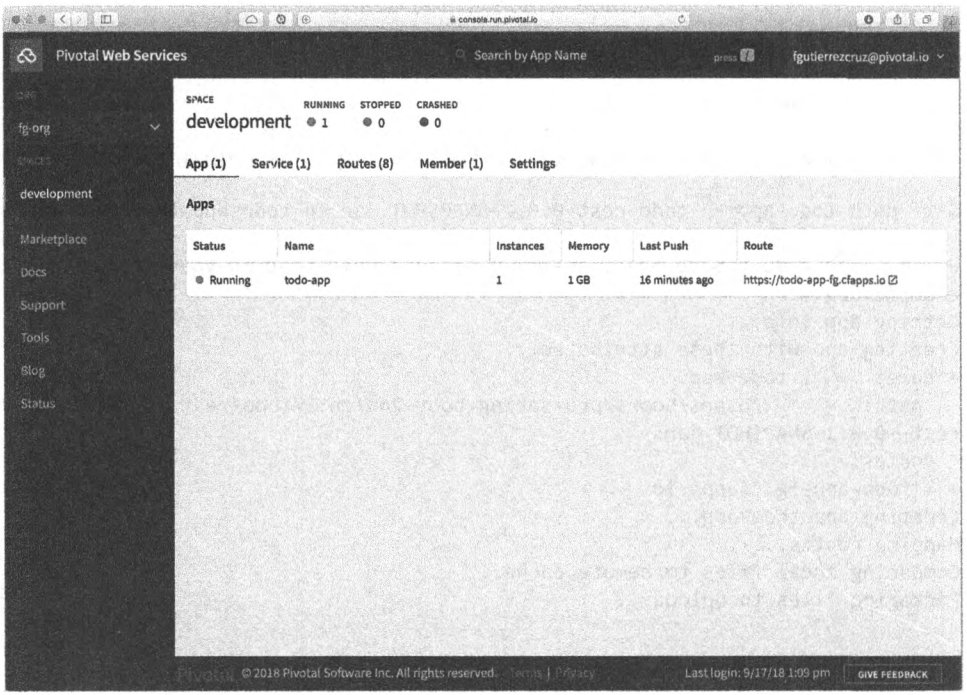


Рис. 12.5. Приложение ToDo в PWS

Если навести указатель мыши на имя приложения todo-app[1], можно увидеть показанное на рис. 12.6.

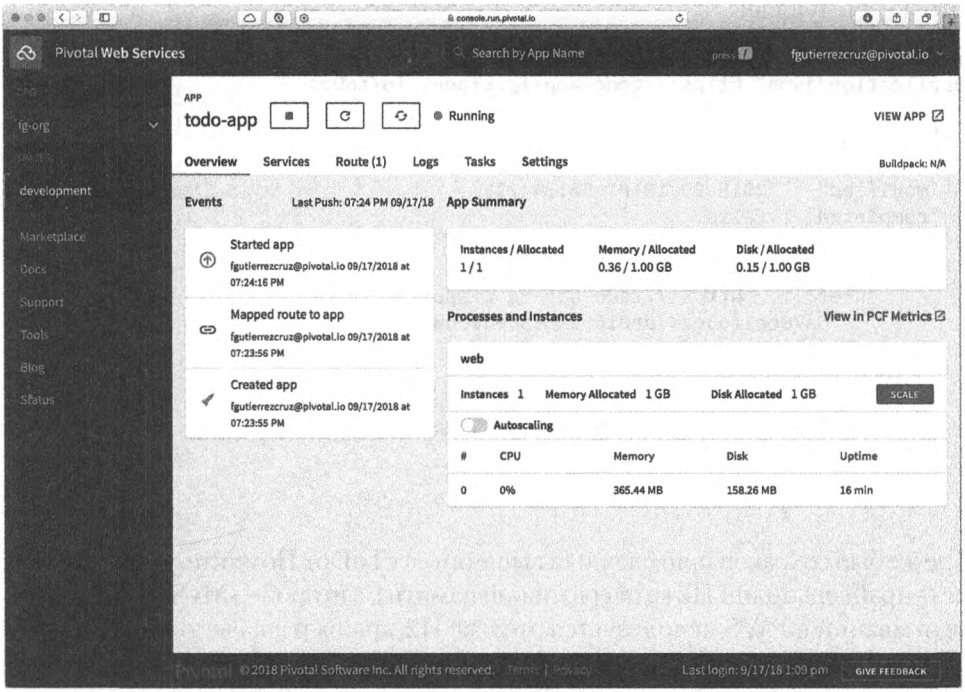


Рис. 12.6. Подробная информация о приложении ToDo PWS

Перейдите по каждой из ссылок. Можете посмотреть на журнал, щелкнув на вкладке Logs (Логи). Можете получить метрики, щелкнув на View in PCF Metrics (Посмотреть метрики PCF), чтобы получить более подробную информацию об используемой приложением оперативной памяти, числе запросов в минуту, использовании CPU, диска и т. д.

Можно также посмотреть журнал с помощью следующей команды:

```
$ cf logs todo-app
```

Эта команда выводит поступающие записи в журнале. Можете обновить или отправить запрос в приложение, чтобы посмотреть журнал. Это удобный способ отладки приложения.

[1] Интерфейс изменился, и теперь для этого нужно щелкнуть на названии приложения. Приводимая в экранной форме информация также отличается от информации на рис. 12.6.

Создание сервисов

Можно добавить в приложение запланированные дела, выполнив в командной строке следующее:

```
$ curl -XPOST -d '{"description":"Learn to play Guitar"}' -H "Content-Type:
application/json" https://todo-app-fg.cfapps.io/toDos
{
  "description" : "Learn to play Guitar",
  "created" : "2018-09-18T01:58:34.211",
  "modified" : "2018-09-18T01:58:34.211",
  "completed" : false,
  "_links" : {
    "self" : {
      "href" :  "https://todo-app-fg.cfapps.io/toDos/
          8a70ee1f65ea47de0165ea668de30000"
    },
    "toDo" : {
      "href" :  "https://todo-app-fg.cfapps.io/toDos/
          8a70ee1f65ea47de0165ea668de30000"
    }
  }
}
```

Где же был сохранен приведенный выше объект ToDo? Помните, у приложения есть драйверы: один H2 (в оперативной памяти), а второй — MySQL? При развертывании в PWS используется этот же H2-драйвер в качестве локального. Почему? А потому, что мы не указали никакого внешнего MySQL-сервиса.

PAS предоставляет возможность создания сервисов. Если вы заглянете снова в раздел, посвященный принципам 12 факторов, то обнаружите там пункт о том, что сервисы следует рассматривать как подключаемые ресурсы. PAS упрощает эту задачу, предоставляя готовые сервисы, которые не нужно устанавливать и которыми не нужно управлять. В терминологии PAS такие сервисы называются *управляемыми сервисами* (managed services).

Взглянем, сколько сервисов есть в PAS. Выполните следующую команду:

```
$ cf marketplace
```

Эта команда выводит список всех доступных управляемых сервисов, заранее установленных и введенных PAS в действие. В данном случае мы воспользуемся сервисом ClearDB, включающим сервис MySQL.

Чтобы запросить у PAS создание экземпляра сервиса `cleardb`, необходимо выполнить команду:

```
$ cf create-service <поставщик> <план> <название_сервиса>
```

Выполните следующую команду для использования MySQL сервиса:

```
$ cf create-service cleardb spark mysql
Creating service instance mysql in org your-org / space development as
your-email@example.org...
OK
```

Мы выбрали бесплатный тарифный план `spark`. Чтобы выбрать другой, необходимо добавить платежную карту, с которой каждый месяц будет списываться определенная сумма денег.

Можно просмотреть используемые сервисы с помощью следующей команды:

```
$ cf services
Getting services in org your-org / space development as your -email@
example.org...

name    service   plan    bound apps   last operation
mysql   cleardb   spark                create succeeded
```

Обратите внимание, что в выводе предыдущей команды столбец `bound apps` (привязанные приложения) пуст. Нам нужно указать приложению ToDo использовать этот сервис (`mysql`). Для привязки приложения к сервису можно выполнить следующую команду:

```
$ cf bind-service todo-app mysql
Binding service mysql to app todo-app in org your-org / space development
as your-email@example.org...
OK
TIP: Use 'cf restage todo-app' to ensure your env variable changes take
effect
```

При этом контейнер, в котором запущено приложение ToDo, создает «за кулисами» переменную среды `VCAP_SERVICES;`, содержащую все учетные данные, которая упрощает для приложения ToDo подключение к сервису `mysql`. Чтобы приложение ToDo увидело эту переменную среды, необходимо его перезапустить с помощью следующей команды:

```
$ cf restart todo-app
```

После перезапуска приложения проверьте, работает ли оно. Перейдите в браузер и добавьте ToDo. Взглянем на переменную `VCAP_SERVICES`. Выполните следующую команду:

```
$ cf env todo-app
Getting env variables for app todo-app in org your-org / space development
as your-email@example.org...
OK
System-Provided:
```

```
{
 "VCAP_SERVICES": {
  "cleardb": [
   {
    "binding_name": null,
    "credentials": {
     "hostname": "us-cdbr-iron-east-01.cleardb.net",
      "jdbcUrl": "jdbc:mysql://us-cdbr-iron-east-01.cleardb.net/
          ad_9a533ebf2e8e79a?user=b2c041b9ef8f25\u0026password=30e7a38b",
     "name": "ad_9a533ebf2e8e79a",
     "password": "30e7a38b",
     "port": "3306",
      "uri":  "mysql://b2c041b9ef8f25:30e7a38b@us-cdbr-iron-east-01.cleardb.
          net:3306/ad_9a533ebf2e8e79a?reconnect=true",
     "username": "b2c041b9ef8f25"
    },
    "instance_name": "mysql",
    "label": "cleardb",
    "name": "mysql",
    "plan": "spark",...
...
....
```

Как видите, переменная `VCAP_SERVICES` содержит свойства `hostname`, `username`, `password` и `jdbcUrl`. С помощью этих данных на самом деле можно выполнить подключение. Можете воспользоваться любым клиентом MySQL и подключиться с помощью этих свойств. Например, при применении утилиты командной строки `mysql` можно выполнить следующее:

```
$ mysql -h us-cdbr-iron-east-01.cleardb.net -ub2c041b9ef8f25 -p30e7a38b
ad_9a533ebf2e8e79a
...
...
mysql> show tables;
+------------------------------+
| Tables_in_ad_9a533ebf2e8e79a |
+------------------------------+
| to_do                        |
+------------------------------+
1 row in set (0.07 sec)
mysql> select * from to_do \G
*************************** 1. row ***************************
         id: 8a72072165ea86ef0165ea887cd10000
  completed:
    created: 2018-09-18 02:35:38
description: Learn to play Guitar
   modified: 2018-09-18 02:35:38
1 row in set (0.07 sec)
mysql>
```

Как вы видите, теперь приложение ToDo использует сервис MySQL. Но как? Cloud Foundry задействует так называемые buildpacks, которые анализируют приложение и определяют, код на каком языке программирования запускается. Cloud Foundry работает независимо от языка программирования; так, в данном случае Cloud Foundry определяет, что используется приложение Spring Boot. Cloud Foundry также обнаруживает ограниченный сервис (`mysql`) и проверяет наличие подходящих драйверов (в данном случае включенного в JAR-файл драйвера MySQL) для подключения. А знаете, что лучше всего? То, что даже не нужно ничего менять в коде! Cloud Foundry и buildpack для Java автоматически настраивают для вас источник данных. Вот так все просто!

Благодаря Cloud Foundry вы можете сосредоточить внимание на своем приложении и не задумываться об инфраструктуре, сервисах и многом другом.

Все, что мы проделали для создания сервиса, можно было сделать и с помощью веб-консоли. Можно просто зайти на вкладку Marketplace (Торговая площадка) на странице PWS и выбрать нужный сервис (рис. 12.7).

Можно щелкнуть на пиктограмме ClearDB MySQL Database (Очистить базу данных MySQL) и выбрать для нее план Spark.

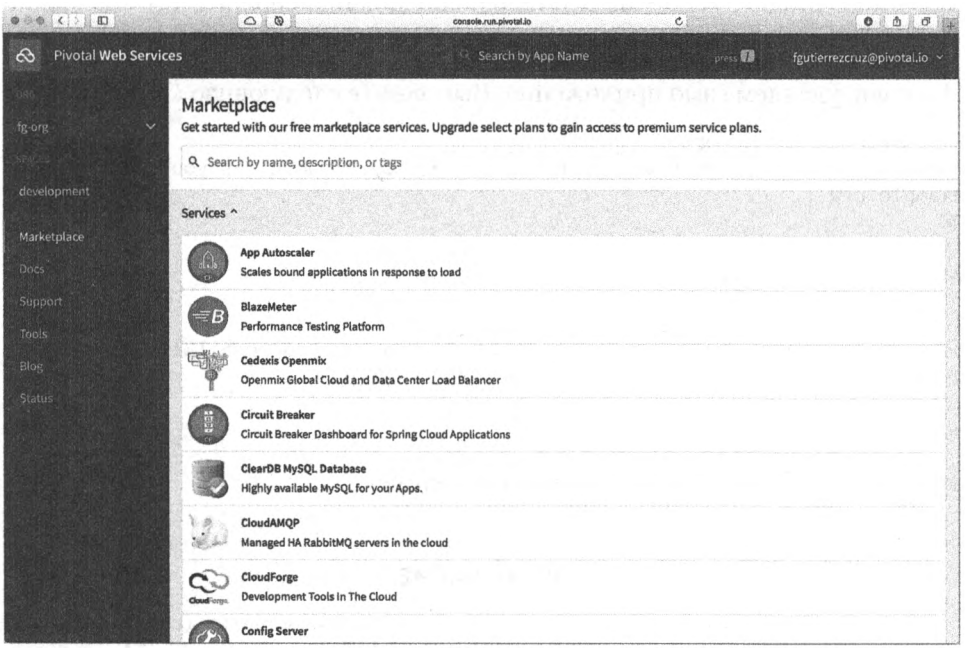


Рис. 12.7. PWS Marketplace

Поздравляем! Теперь вы умеете развертывать приложения Spring Boot в облаке. Сумеете развернуть остальные наши примеры? Как насчет проектов `todo-mongo`, `todo-redis`, `todo-rabbitmq`?

Убираем за собой

Важно удалить свои сервисы и приложения. Благодаря этому при необходимости вам будет доступно больше ресурсов. Начнем с отвязки сервиса от приложения. Выполните следующую команду:

```
$ cf unbind-service todo-app mysql
Unbinding app todo-app from service mysql in org your-org / space
development as your-email@example.org...
OK
```

Удалим этот сервис с помощью следующей команды:

```
$ cf delete-service mysql
Really delete the service mysql?> y
Deleting service mysql in org your-org / space development as your-email@
exmaple.org...
OK
```

Как вы видите, необходимо подтвердить удаление сервиса. Во избежание этого можно использовать флаг `-f`.

Наконец, удаляем само приложение. Выполните следующую команду:

```
$ cf delete -f todo-app
Deleting app todo-app in org your-org / space development as your-email@
example.org...
OK
```

Можете выполнить:

```
$ cf apps
Getting apps in org your-org / space development as your-email@example.org...
OK

No apps found
```

чтобы просмотреть список своих приложений.

ПРИМЕЧАНИЕ

Напомню, что исходный код для данной книги можно скачать на сайте Apress или из репозитория GitHub: https://github.com/Apress/pro-spring-boot-2.

Резюме

В этой главе вы узнали больше о микросервисах и облачных приложениях. Вы также узнали о принципах 12 факторов, благодаря которым можно создавать нативные облачные приложения.

Я также рассказал вам о платформе Cloud Foundry и ее возможностях, включая сервис приложений Pivotal и сервис контейнеров Pivotal. Вы теперь знаете, что Cloud Foundry отличается независимостью от языка программирования, а buildpacks анализируют приложение и автоматически производят нужные настройки.

В следующей главе вам предстоит узнать, как расширять модули Spring Boot и создавать свои собственные.

13 Расширение возможностей Spring Boot

Разработчики и архитекторы программного обеспечения находятся в постоянном поиске подходящих паттернов проектирования, новых алгоритмов для реализации, удобных для применения и поддержки переиспользуемых компонентов, а также новых способов усовершенствования разработки. Найти единственное идеальное решение далеко не всегда просто, так что приходится использовать различные технологии и методологии для создания приложений, которые бы работали без каких-либо сбоев.

В этой главе рассказывается о созданном командами Spring и Spring Boot паттерне переиспользования компонентов, которые легко применять и реализовывать. На самом деле мы говорили об этом паттерне на протяжении всей книги, особенно в главе о конфигурации Spring Boot.

В этой главе подробно описана автоконфигурация, включая способы расширения старых и создания новых, переиспользуемых, модулей Spring Boot. Приступим.

Создание spring-boot-starter

В этом разделе я покажу вам, как создать пользовательский `spring-boot-starter`, но сначала обсудим некоторые требования. Поскольку мы работаем в приложении ToDo, этот «стартовый пакет» для приложения будет представлять собой клиент, с помощью которого можно выполнять различные

операции над объектами ToDo, например create, find и findAll. Для этого клиента нужен хост с возможностью подключения к сервису REST API ToDo.

Начнем с создания проекта. Пока не существует эталонного шаблона для создания пользовательского spring-boot-starter, так что придется сделать это вручную. Создайте следующую структуру каталогов:

```
todo-client-starter/
├── todo-client-spring-boot-autoconfigure
└── todo-client-spring-boot-starter
```

Необходимо создать каталог todo-client-starter, а внутри него — два подкаталога: todo-client-spring-boot-autoconfigure и todo-client-spring-boot-starter. Да, существует определенное соглашение о наименованиях. Команда создателей Spring Boot предлагает следующее соглашение о наименованиях для всех пользовательских «стартовых пакетов»: <название_стартового_пакета>-spring-boot-starter и <название_стартового_пакета>-spring-boot-autoconfigure. В модуле autoconfigure содержится весь необходимый для «стартового пакета» код и зависимости; не беспокойтесь, я расскажу вам, что именно нужно.

Во-первых, создайте основной файл pom.xml с описанием двух модулей: autoconfigure и starter. Создайте файл pom.xml в каталоге todo-client-starter. Структура каталогов должна выглядеть следующим образом:

```
todo-client-starter/
├── pom.xml
├── todo-client-spring-boot-autoconfigure
└── todo-client-spring-boot-starter
```

Файл pom.xml должен выглядеть так, как показано в листинге 13.1.

Листинг 13.1. todo-client-starter/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://
             www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
             http://maven.apache.org/xsd/maven-4.0.0.xsd">
      <modelVersion>4.0.0</modelVersion>
      <groupId>com.apress.todo</groupId>
      <artifactId>todo-client</artifactId>
      <version>0.0.1-SNAPSHOT</version>
      <packaging>pom</packaging>
      <name>todo-client</name>

      <modules>
            <module>todo-client-spring-boot-autoconfigure</module>
            <module>todo-client-spring-boot-starter</module>
```

```
    </modules>
    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-dependencies</artifactId>
                <version>2.0.5.RELEASE</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>
</project>
```

В листинге 13.1 приведен основной файл `pom.xml` с двумя модулями. Важно отметить, что тег `<packaging/>` содержит значение `pom`, поскольку в итоге нужно будет установить эти JAR-файлы в локальный репозиторий для дальнейшего использования. Обратите также внимание, что в этом файле `pom.xml` объявлен тег `<dependencyManagement/>`, который позволяет использовать JAR-файлы Spring Boot и все его зависимости. И наконец, объявлять версии не нужно.

Модуль todo-client-spring-boot-starter

Создайте еще один файл `pom.xml` в каталоге `todo-client-spring-boot-starter`. Структура каталогов должна оказаться следующей:

```
todo-client-starter/
├── pom.xml
├── todo-client-spring-boot-autoconfigure
└── todo-client-spring-boot-starter
    └── pom.xml
```

См. листинг 13.2.

Листинг 13.2. todo-client-starter/todo-client-spring-boot-starter/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.todo</groupId>
    <artifactId>todo-client-spring-boot-starter</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>
```

```
    <name>todo-client-spring-boot-starter</name>
    <description>Todo Client Spring Boot Starter</description>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
         <project.reporting.outputEncoding>UTF-8</project.reporting.
             outputEncoding>
    </properties>

 <parent>
        <groupId>com.apress.todo</groupId>
        <artifactId>todo-client</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <relativePath>..</relativePath>
    </parent>

 <dependencies>
        <dependency>
            <groupId>com.apress.todo</groupId>
            <artifactId>todo-client-spring-boot-autoconfigure</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
    </dependencies>

</project>
```

Ничего нового. Описывается тег `<parent/>`, относящийся к предыдущему файлу `pom.xml`, а также описывается модуль `autoconfigure`.

Вот и весь модуль `todo-client-spring-boot-starter`; больше ничего в нем нет. Его можно рассматривать просто как место объявления модулей, которые будут, собственно, выполнять всю работу.

Модуль todo-client-spring-boot-autoconfigure

Создадим структуру внутри каталога `todo-client-spring-boot-autoconfigure`. Итоговая структура должна выглядеть следующим образом:

```
todo-client-starter/
├── pom.xml
├── todo-client-spring-boot-autoconfigure
│   ├── pom.xml
│   └── src
│       └── main
│           ├── java
│           └── resources
└── todo-client-spring-boot-starter
    └── pom.xml
```

Каталог `todo-client-spring-boot-autoconfigure` должен выглядеть вот так:

```
todo-client-spring-boot-autoconfigure/
├── pom.xml
└── src
    └── main
        ├── java
        └── resources
```

Базовая структура проекта Java. Начнем с файла `pom.xml` (листинг 13.3).

Листинг 13.3. todo-client-starter/todo-client-spring-boot-autoconfigure/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.todo</groupId>
    <artifactId>todo-client-spring-boot-autoconfigure</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>8</source>
                    <target>8</target>
                </configuration>
            </plugin>
        </plugins>
    </build>
    <packaging>jar</packaging>

    <name>todo-client-spring-boot-autoconfigure</name>
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
         <project.reporting.outputEncoding>UTF-8</project.reporting.
             outputEncoding>
        <java.version>1.8</java.version>
    </properties>

    <parent>
        <groupId>com.apress.todo</groupId>
        <artifactId>todo-client</artifactId>
        <version>0.0.1-SNAPSHOT</version>
```

```
    <relativePath>..</relativePath>
</parent>

<dependencies>

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

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

    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>1.18.2</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.hateoas</groupId>
        <artifactId>spring-hateoas</artifactId>
    </dependency>

    <!-- JSON / Traverson -->
    <dependency>
        <groupId>org.springframework.plugin</groupId>
        <artifactId>spring-plugin-core</artifactId>
    </dependency>

    <dependency>
        <groupId>com.jayway.jsonpath</groupId>
        <artifactId>json-path</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-configuration-processor</artifactId>
        <optional>true</optional>
    </dependency>

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

```
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

</project>
```

В нашем случае проект `autoconfigure` зависит от пакетов Web, Lombok, Security, HateOAS и JsonPath.

Файл spring.factories

Если вы помните из первых глав, я рассказывал, что Spring Boot производит автоконфигурацию на основе пути к классам; в этом его истинная магия. Я упоминал, что при запуске приложения автоконфигурация Spring Boot загружает все классы автоконфигураций из файла `META-INF/spring.factories`, благодаря чему приложение получает все необходимое для работы. Не забывайте, что Spring Boot — *продуманная среда выполнения* для приложений Spring.

Создадим файл `spring.factories` и укажем в нем класс, выполняющий автоматическую конфигурацию и задание некоторых значений по умолчанию (листинг 13.4).

Листинг 13.4. src/main/resources/META-INF/spring.factories

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.apress.
todo.configuration.ToDoClientAutoConfiguration
```

Обратите внимание, что в файле `spring.factories` объявлен класс `ToDoClientAutoConfiguration`.

Автоконфигурация

Начнем с создания класса `ToDoClientAutoConfiguration` (листинг 13.5).

Листинг 13.5. com.apress.todo.configuration.ToDoClientAutoConfiguration.java

```
package com.apress.todo.configuration;

import com.apress.todo.client.ToDoClient;
import lombok.RequiredArgsConstructor;
import org.slf4j.Logger;
```

```
import org.slf4j.LoggerFactory;
import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.context.properties.
  EnableConfigurationProperties;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.hateoas.Resource;
import org.springframework.web.client.RestTemplate;

@RequiredArgsConstructor
@Configuration
@ConditionalOnClass({Resource.class, RestTemplateBuilder.class})
@EnableConfigurationProperties(ToDoClientProperties.class)
public class ToDoClientAutoConfiguration {

    private final Logger log = LoggerFactory.getLogger
        (ToDoClientAutoConfiguration.class);
    private final ToDoClientProperties toDoClientProperties;

    @Bean
    public ToDoClient client(){
        log.info(">>> Creating a ToDo Client...");
        return new ToDoClient(new RestTemplate(),this.toDoClientProperties);
    }
}
```

В листинге 13.5 приведена выполняемая автоконфигурация. Класс снабжен аннотацией `@ConditionalOnClass`, означающей, что конфигурация будет выполняться только при наличии классов `Resource.class` и `RestTemplateBuilder.class` по пути к классам. Конечно, поскольку одна из наших зависимостей — `spring-boot-starter-web`, эти классы присутствуют. Но что будет, если их исключить? Тогда-то аннотация и сработает.

В этом классе объявляется компонент `ToDoClient`, использующий экземпляры `RestTemplate` и `ToDoClientProperties`.

Вот и все. Чрезвычайно простая автоконфигурация. Если указанные классы ресурсов найдены в проекте, использующем данный «стартовый пакет», задается применяемый по умолчанию компонент `ToDoClient`.

Вспомогательные классы

Далее создадим вспомогательные классы. Создайте классы `ToDoClientProperties` и `ToDoClient` (листинги 13.6 и 13.7).

Листинг 13.6. com.apress.todo.configuration.ToDoClientProperties.java

```
package com.apress.todo.configuration;

import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;

@Data
@ConfigurationProperties(prefix="todo.client")
public class ToDoClientProperties {

    private String host = "http://localhost:8080";
    private String path = "/toDos";

}
```

Как видите, ничего нового — просто два поля со значениями по умолчанию для хоста и пути. Это значит, что можно их переопределить в файлах `application.properties`.

Листинг 13.7. com.apress.todo.client.ToDoClient.java

```
package com.apress.todo.client;

import com.apress.todo.configuration.ToDoClientProperties;
import com.apress.todo.domain.ToDo;
import lombok.AllArgsConstructor;
import lombok.Data;
import org.springframework.core.ParameterizedTypeReference;
import org.springframework.hateoas.MediaTypes;
import org.springframework.hateoas.Resources;
import org.springframework.hateoas.client.Traverson;
import org.springframework.http.MediaType;
import org.springframework.http.RequestEntity;
import org.springframework.http.ResponseEntity;
import org.springframework.web.client.RestTemplate;
import org.springframework.web.util.UriComponents;
import org.springframework.web.util.UriComponentsBuilder;

import java.net.URI;
import java.util.Collection;

@AllArgsConstructor
@Data
public class ToDoClient {

    private RestTemplate restTemplate;
    private ToDoClientProperties props;
```

```
    public ToDo add(ToDo toDo){
        UriComponents uriComponents = UriComponentsBuilder.newInstance()
                .uri(URI.create(this.props.getHost())).path(this.props.
                getPath()).build();

        ResponseEntity<ToDo> response =
                this.restTemplate.exchange(
                       RequestEntity.post(uriComponents.toUri())
                               .body(toDo)
                       ,new ParameterizedTypeReference<ToDo>() {});
        return response.getBody();
    }

    public ToDo findById(String id){
        UriComponents uriComponents = UriComponentsBuilder.newInstance().
                                uri(URI.create(this.props.getHost())).
                                pathSegment(this.props.getPath(), "/{id}").
                                buildAndExpand(id);

        ResponseEntity<ToDo> response =
                this.restTemplate.exchange(
                         RequestEntity.get(uriComponents.toUri()).
                         accept(MediaTypes.HAL_JSON).build()
                        ,new ParameterizedTypeReference<ToDo>() {});

        return response.getBody();
    }

    public Collection<ToDo> findAll() {
        UriComponents uriComponents = UriComponentsBuilder.newInstance()
                .uri(URI.create(this.props.getHost())).build();

        Traverson traverson = new Traverson(uriComponents.toUri(),
        MediaTypes.HAL_JSON, MediaType.APPLICATION_JSON_UTF8);
        Traverson.TraversalBuilder tb = traverson.follow(this.props.
          getPath().substring(1));
        ParameterizedTypeReference<Resources<ToDo>> typeRefDevices =
          new ParameterizedTypeReference<Resources<ToDo>>() {};

        Resources<ToDo> toDos = tb.toObject(typeRefDevices);

        return toDos.getContent();
    }
}
```

Реализация класса `ToDoClient` очень проста. Во всех методах этого класса применяется `RestTemplate`; и хотя метод `findAll` использует экземпляр *Traverson*

(Java-реализацию JavaScript-библиотеки Traverson (https://github.com/traverson/traverson) для работы со всеми ссылками HATEOAS), но «за кулисами» все равно используется `RestTemplate`.

Уделите немного времени анализу кода. Учтите, что этот клиент выполняет запросы и операции `POST` к серверу приложения REST API ToDo.

Для использования данного клиента необходимо создать класс предметной области ToDo (листинг 13.8).

Листинг 13.8. com.apress.todo.domain.ToDo.java

```
package com.apress.todo.domain;
import com.fasterxml.jackson.annotation.JsonFormat;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.databind.annotation.JsonDeserialize;
import com.fasterxml.jackson.datatype.jsr310.deser.LocalDateTimeDeserializer;
import lombok.Data;
import lombok.NoArgsConstructor;

import java.time.LocalDateTime;

@JsonIgnoreProperties(ignoreUnknown = true)
@NoArgsConstructor
@Data
public class ToDo {

    private String id;
    private String description;

    @JsonDeserialize(using = LocalDateTimeDeserializer.class)
    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private LocalDateTime created;

    @JsonDeserialize(using = LocalDateTimeDeserializer.class)
    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private LocalDateTime modified;

    private boolean completed;

    public ToDo(String description){
        this.description = description;
    }
}
```

Здесь появляются аннотации `@Json*`. Одна из них служит для игнорирования всех ссылок (из протокола HAL+JSON), а другая сериализует экземпляры `LocalDateTime`.

Почти готово. Добавим утилиту для обеспечения безопасности, которая будет зашифровывать/расшифровывать описания запланированных дел.

Создание функциональности @Enable*

Одна из замечательных возможностей технологий Spring и Spring Boot – предоставление ими нескольких функциональностей `@Enable*`, скрывающих весь стереотипный код конфигурации и выполняющих за нас всю грязную работу.

Создадим пользовательскую функциональность `@EnableToDoSecurity`. Начнем с создания аннотации автоконфигурации, которую мог бы обнаружить Spring Boot (листинг 13.9).

Листинг 13.9. com.apress.todo.annotation.EnableToDoSecurity.java

```
package com.apress.todo.annotation;

import com.apress.todo.security.ToDoSecurityConfiguration;
import org.springframework.context.annotation.Import;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Import(ToDoSecurityConfiguration.class)
public @interface EnableToDoSecurity {
    Algorithm algorithm() default Algorithm.BCRYPT;
}
```

В этой аннотации используется перечисляемый тип `Algorithm`; создадим его (листинг 13.10).

Листинг 13.10. com.apress.todo.annotation.Algorithm.java

```
package com.apress.todo.annotation;

public enum Algorithm {
    BCRYPT, PBKDF2
}
```

Это значит, что мы можем передать аннотации `@EnableToDoSecurity` нужные нам параметры. Возможны два значения: `BCRYPT` и `PBKDF2`, и если параметр не указан, то по умолчанию используется `BCRYPT`.

Создайте класс `ToDoSecurityConfiguration`, который запускает конфигурацию, если объявлена аннотация `@EnableToDoSecurity` (листинг 13.11).

Листинг 13.11. com.apress.todo.security.ToDoSecurityConfiguration.java

```
package com.apress.todo.security;

import com.apress.todo.annotation.Algorithm;
import com.apress.todo.annotation.EnableToDoSecurity;
import org.springframework.context.annotation.ImportSelector;
import org.springframework.core.annotation.AnnotationAttributes;
import org.springframework.core.type.AnnotationMetadata;

public class ToDoSecurityConfiguration implements ImportSelector {
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        AnnotationAttributes attributes =
                AnnotationAttributes.fromMap(
                        annotationMetadata.getAnnotationAttributes
                        (EnableToDoSecurity.class.getName(), false));
        Algorithm algorithm = attributes.getEnum("algorithm");
        switch(algorithm){
            case PBKDF2:
                return new String[] {"com.apress.todo.security.
                    Pbkdf2Encoder"};
            case BCRYPT:
            default:
                return new String[] {"com.apress.todo.security.
                    BCryptEncoder"};
        }
    }
}
```

Листинг 13.11 демонстрирует, что автоконфигурация выполняется, только если объявлена аннотация `@EnableToDoSecurity`. Spring Boot также отслежи-

вает все классы, реализующие интерфейс `ImportSelector`, скрывающий весь стереотипный код обработки аннотаций.

Итак, если обнаружена аннотация `@EnableToDoSecurity`, то этот класс выполняется путем вызова метода `selectImports`, возвращающего массив строковых значений, соответствующих классам, которые необходимо настроить, в данном случае либо классу `com.apress.todo.security.Pbkdf2Encoder` (если в качестве параметра передан алгоритм `PBKDF2`), либо классу `com.apress.todo.security.BCryptEncoder` (если передан `BCRYPT`).

Что такого особенного в этих классах? Создадим классы `BCryptEncoder` и `Pbkdf2Encoder` (листинги 13.12 и 13.13).

Листинг 13.12. com.apress.todo.security.BCryptEncoder.java

```
package com.apress.todo.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
public class BCryptEncoder {

    @Bean
    public ToDoSecurity utils(){
        return new ToDoSecurity(new BCryptPasswordEncoder(16));
    }
}
```

Листинг 13.13. com.apress.todo.security.Pbkdf2Encoder.java

```
package com.apress.todo.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.password.Pbkdf2PasswordEncoder;

@Configuration
public class Pbkdf2Encoder {

    @Bean
    public ToDoSecurity utils(){
        return new ToDoSecurity(new Pbkdf2PasswordEncoder());
    }
}
```

В обоих классах объявлен компонент `ToDoSecurity`. Так, если выбрать алгоритм `PBKDF2`, то компонент `ToDoSecurity` будет создан с экземпляром `Pbkdf2PasswordEncoder`; а если алгоритм `BCRYPT`, то компонент `ToDoSecurity` будет создан с экземпляром `BCryptPasswordEncoder(16)`.

В листинге 13.14 приведен класс `ToDoSecurity`.

Листинг 13.14. com.apress.todo.security.ToDoSecurity.java

```
package com.apress.todo.security;

import lombok.AllArgsConstructor;
import lombok.Data;
import org.springframework.security.crypto.password.PasswordEncoder;

@AllArgsConstructor
@Data
public class ToDoSecurity {
    private PasswordEncoder encoder;
}
```

Как видите, ничего особенного в этом классе нет.

Сервис REST API приложения ToDo

Подготовим сервис REST API приложения ToDo. Можете переиспользовать проект `todo-rest`, применяющий созданные в предыдущих главах `data-jpa` и `data-rest`. Взглянем на него снова и определим, что нужно сделать (листинги 13.15–13.17).

Листинг 13.15. com.apress.todo.domain.ToDo.java

```
package com.apress.todo.domain;

import com.fasterxml.jackson.annotation.JsonFormat;
import com.fasterxml.jackson.databind.annotation.JsonDeserialize;
import com.fasterxml.jackson.datatype.jsr310.deser.LocalDateTimeDeserializer;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.annotations.GenericGenerator;

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

```
import javax.persistence.PrePersist;
import javax.persistence.PreUpdate;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.NotNull;
import java.time.LocalDateTime;

@Entity
@Data
@NoArgsConstructor
public class ToDo {

    @NotNull
    @Id
    @GeneratedValue(generator = "system-uuid")
    @GenericGenerator(name = "system-uuid", strategy = "uuid")
    private String id;
    @NotNull
    @NotBlank
    private String description;

    @Column(insertable = true, updatable = false)
    @JsonDeserialize(using = LocalDateTimeDeserializer.class)
    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private LocalDateTime created;

    @JsonDeserialize(using = LocalDateTimeDeserializer.class)
    @JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")
    private LocalDateTime modified;

    private boolean completed;

    public ToDo(String description){
        this.description = description;
    }

    @PrePersist
    void onCreate() {
        this.setCreated(LocalDateTime.now());
        this.setModified(LocalDateTime.now());
    }

    @PreUpdate
    void onUpdate() {
        this.setModified(LocalDateTime.now());
    }
}
```

В этом классе `ToDo` нет ничего нового; все используемые здесь аннотации вам уже хорошо знакомы. Единственное отличие — аннотации `@Json*` применяются только для дат в конкретном формате.

Листинг 13.16. com.apress.todo.repository.ToDoRepository.java

```
package com.apress.todo.repository;

import com.apress.todo.domain.ToDo;
import org.springframework.data.repository.CrudRepository;

public interface ToDoRepository extends CrudRepository<ToDo,String> {

}
```

Такой же, как и раньше; ничего, что бы вы еще об этом интерфейсе не знали.

Листинг 13.17. com.apress.todo.config.ToDoRestConfig.java

```
package com.apress.todo.config;

import com.apress.todo.domain.ToDo;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.rest.core.config.RepositoryRestConfiguration;
import org.springframework.data.rest.webmvc.config.
  RepositoryRestConfigurerAdapter;

@Configuration
public class ToDoRestConfig extends RepositoryRestConfigurerAdapter {

    @Override
    public void configureRepositoryRestConfiguration
        (RepositoryRestConfiguration config) {
        config.exposeIdsFor(ToDo.class);
    }
}
```

В листинге 13.17 приведен новый класс `ToDoRestConfig`, расширяющий класс `RepositoryRestConfigurerAdapter`; этот класс помогает в настройке части реализации `RestController` на основе настроек по умолчанию, выполняемых автоконфигурацией репозиториев JPA. В нем переопределяется метод `configureRepositoryRestConfiguration`, обеспечивающий доступ к идентификаторам класса домена. Когда мы работали с REST в других главах, идентификаторы никогда не выдавались по запросу; теперь же это реально. И такая возможность нам необходима, чтобы получить идентификатор экземпляра ToDo.

В файле `application.properties` должно находиться следующее:

```
# JPA
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```

Опять же ничего для вас нового.

Установка и тестирование

Подготовим все необходимое для работы с новым пользовательским «стартовым пакетом». Начнем с установки `todo-client-spring-boot-starter`. Откройте терминал, перейдите в каталог `todo-client-starter` и выполните следующую команду:

```
$ mvn clean install
```

Эта команда устанавливает наш JAR-файл в локальный каталог `.m2`, где его смогут найти другие использующие его проекты.

Проект Task

После установки `todo-client-spring-boot-starter` можно его протестировать. Вам нужно создать новый проект. Можете сделать это как обычно. Перейдите в Spring Initializr (https://start.spring.io) и внесите следующие значения в соответствующие поля.

- Group (Группа): `com.apress.task`.
- Artifact (Артефакт): `task`.
- Name (Название): `task`.
- Package Name (Название пакета): `com.apress.task`.

Можете выбрать в качестве типа проекта Maven или Gradle. Затем нажмите кнопку Generate Project (Сгенерировать проект) и скачайте ZIP-файл. Разархивируйте его и импортируйте в предпочитаемую вами интегрированную среду разработки (рис. 13.1).

Далее необходимо добавить `todo-client-spring-boot-starter`. Если вы используете Maven, перейдите в файл `pom.xml` и добавьте следующую зависимость:

```
<dependency>
    <groupId>com.apress.todo</groupId>
    <artifactId>todo-client-spring-boot-starter</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

Если вы используете Gradle, добавьте в файл `build.gradle` следующую зависимость:

```
compile('com.apress.todo:todo-client-spring-boot-starter:0.0.1-SNAPSHOT')
```

Вот и все. Теперь откройте основной класс, в котором содержится код, приведенный в листинге 13.18.

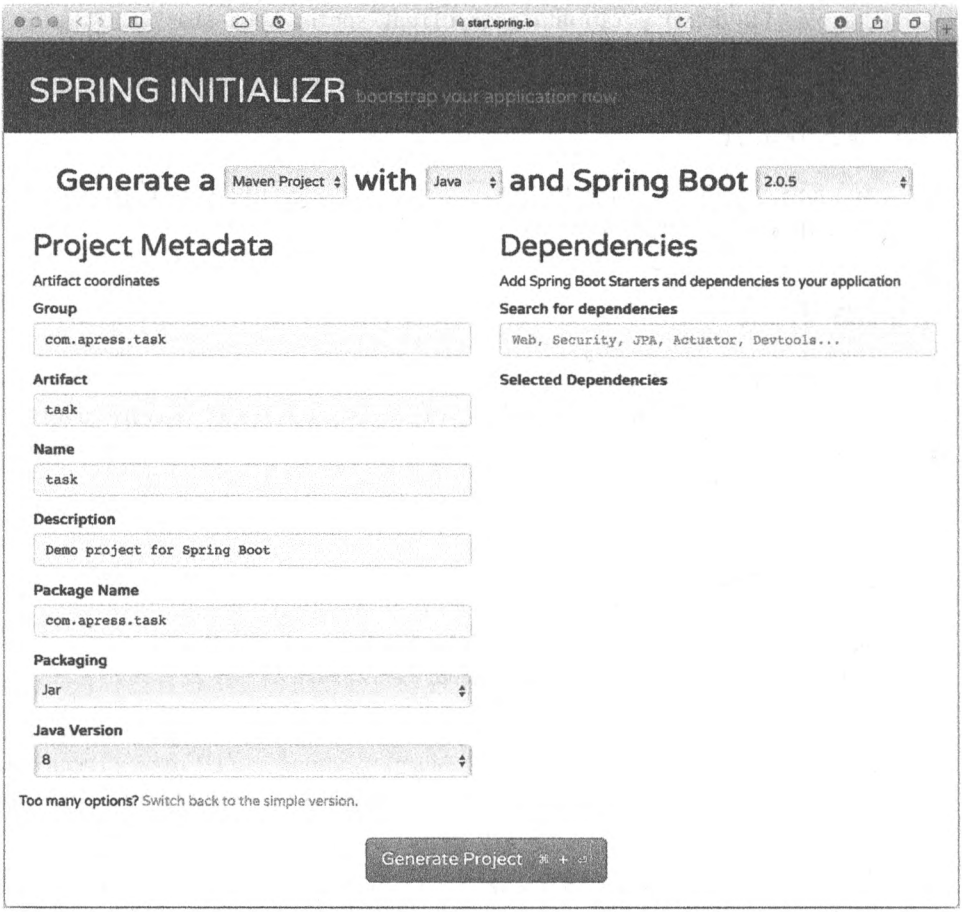


Рис. 13.1. Spring Initializr

Листинг 13.18. com.apress.task.TaskApplication.java

```
package com.apress.task;

import com.apress.todo.annotation.EnableToDoSecurity;
import com.apress.todo.client.ToDoClient;
import com.apress.todo.domain.ToDo;
import com.apress.todo.security.ToDoSecurity;
import org.springframework.boot.ApplicationRunner;
import org.springframework.boot.SpringApplication;
```

```
import org.springframework.boot.WebApplicationType;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;

@EnableToDoSecurity
@SpringBootApplication
public class TaskApplication {

      public static void main(String[] args) {
        SpringApplication app = new SpringApplication(TaskApplication.class);
        app.setWebApplicationType(WebApplicationType.NONE);
        app.run(args);
      }

    @Bean
    ApplicationRunner createToDos(ToDoClient client){
          return args -> {
            ToDo toDo = client.add(new ToDo("Read a Book"));
            ToDo review = client.findById(toDo.getId());
            System.out.println(review);
            System.out.println(client.findAll());
        };
    }

    @Bean
    ApplicationRunner secure(ToDoSecurity utils){
        return args -> {
            String text = "This text will be encrypted";
            String hash = utils.getEncoder().encode(text);
            System.out.println(">>> ENCRYPT: " + hash);
            System.out.println(">>> Verify: " + utils.getEncoder().
                matches(text,hash));
        };
    }
}
```

Здесь содержатся два компонента `ApplicationRunner`, у каждого из которых есть по параметру. `createToDos` использует экземпляр компонента `ToDoClient` (если у вас по пути к классам нет классов `Resource` и `RestTemplateBuilder`, произойдет сбой). Он задействует уже известные вам методы (`add`, `findById` и `findAll`).

Метод `secure` использует экземпляр компонента `ToDoSecurity`, что возможно благодаря аннотации `@EnableToDoSecurity`. Если удалить ее или закомментировать, вы получите сообщение о невозможности найти компонент `ToDoSecurity`.

Уделите немного времени анализу этого кода, чтобы разобраться, что в нем происходит.

Запуск приложения Task

Прежде чем запускать приложение Task, убедитесь сначала, что приложение `todo-rest` запущено. Оно должно работать на порте 8080. Напомню, что вы уже установили `todo-client-spring-boot-starter` с помощью команды `mvn clean install`.

Если вы его запустите, то увидите, что возвращено несколько ответов, а объект ToDo сохранен в сервисе ToDo REST. Будет также отображен зашифрованный текст.

Если API REST приложения ToDo работает у вас на другом IP, хосте, порте или пути, можете поменять значения по умолчанию с помощью свойств `todo.client.*` в файле `application.properties`.

```
# ToDo Rest API
todo.client.host=http://some-new-server.com:9091
todo.client.path=/api/toDos
```

Помните, что, если их не переопределить, по умолчанию будут использоваться значения http://localhost:8080 и /toDos. После запуска приложения Task вы должны увидеть в журнале примерно следующий вывод:

```
INFO - [ main] c.a.t.c.ToDoClientAutoConfiguration      : >>> Creating a ToDo
Client...
INFO - [ main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans
for JMX exposure on startup
INFO - [ main] com.apress.task.TaskApplication          : Started
TaskApplication in 1.047 seconds (JVM running for 1.853)
ToDo(id=8a8080a365f427c00165f42adee50000, description=Read a Book,
created=2018-09-19T17:29:34, modified=2018-09-19T17:29:34, completed=false)
[ToDo(id=8a8080a365f427c00165f42adee50000, description=Read a
Book, created=2018-09-19T17:29:34, modified=2018-09-19T17:29:34,
completed=false)]
>>> ENCRYPT: $2a$16$pVOI../twnLwN3GFiChdR.zRFfyCIZMEbwEXbAtRoIHqxeLB3gmUG
>>> Verify: true
```

Поздравляю! Вы только что создали свой первый пользовательский «стартовый пакет» Spring Boot и функциональность `@Enable`!

ПРИМЕЧАНИЕ

Напомню, что исходный код для данной книги можно скачать на сайте Apress или в репозитории GitHub: https://github.com/Apress/pro-spring-boot-2.

Резюме

В этой главе было показано, как создать модуль для Spring Boot с помощью паттерна автоконфигурации. Я показал вам, как создать свой собственный монитор состояния приложения. Как видите, расширять приложения Spring Boot очень просто, так что не стесняйтесь модифицировать код и экспериментировать с ним.

Мы не проводили практически никакого модульного или интеграционного тестирования. В качестве домашнего задания можете попробовать все изложенное на практике. Уверен, это поможет вам еще лучше понять, как работает Spring Boot. Повторение — мать учения!

Приложение. Интерфейс командной строки Spring Boot

Здесь обсуждается утилита Spring Boot, с помощью которой можно создавать прототипы и готовые к промышленной эксплуатации приложения. Она включена в установочный пакет Spring Boot, знакомый вам по предыдущим главам. Это не зависимость/плагин Maven или Gradle.

Я имею в виду Spring Boot CLI (интерфейс командной строки). В предыдущих главах вы узнали, что CLI можно установить из бинарного дистрибутива, который можно найти по адресу http://repo.spring.io/release/org/springframework/boot/spring-boot-cli/. Или, если вы работаете на Mac/Linux, можете воспользоваться Homebrew (http://brew.sh/) с помощью следующей команды:

```
$ brew tap pivotal/tap
$ brew install springboot
```

Если вы применяете Linux, можете также воспользоваться утилитой SDKMAN (http://sdkman.io/) и установить Spring Boot CLI с помощью следующей команды:

```
$ sdk install springboot
```

Все примеры приведены на языках Java и Groovy. Никаких особых различий между этими языками в смысле компиляции, запуска и компоновки нет. Единственное различие — несколько дополнительных параметров, передаваемых в командной строке. Но не волнуйтесь, я сейчас расскажу вам о них.

Spring Boot CLI

Когда я впервые начал изучать Spring Boot, примерно три года назад, он представлял собой альфа-версию, в которой была доступна только команда `run`. Что еще может быть нужно, правда? Казалось потрясающим, что можно

создать и запустить приложение на Groovy с помощью всего нескольких строк кода. Просто и впечатляюще.

Начиная с версии 1.4, появились дополнительные возможности и интерактивная оболочка, которую вы вскоре увидите. Для подробного изучения CLI нам понадобится несколько простых примеров. Начнем с приведенного в листинге П.1, который демонстрирует пример Groovy из других глав.

Листинг П.1. app.groovy

```
@RestController
class WebApp{

    @RequestMapping("/")
    String greetings(){
        "Spring Boot Rocks"
   }
}
```

В листинге П.1 приведено простейшее веб-приложение Groovy, которое можно запустить с помощью Spring Boot. Взглянем на то же веб-приложение на Java (листинг П.2).

Листинг П.2. WebApp.java

```
package com.apress.spring;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@SpringBootApplication
public class WebApp {

   @RequestMapping("/")
   public String greetings(){
      return "Spring Boot Rocks in Java too!";
   }

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

В листинге П.2 приведена Java-версия того же простейшего веб-приложения. Как я упоминал, Spring Boot позволяет выбрать Java или Groovy в качестве

языка создания корпоративных и готовых к промышленной эксплуатации приложений.

Начнем использовать команды CLI.

Команда run

Команда `run` позволяет запускать Java- или Groovy-приложения Spring Boot. Ее синтаксис приведен ниже:

```
spring run [опции] <файлы> [--] [аргументы]
```

Доступные опции перечислены в табл. П.1.

Таблица П.1. Опции команды spring run

Опция	Описание
--autoconfigure [Boolean]	Добавляет автоконфигурацию преобразований компилятора. Запоминает возможности автоконфигурации и нюансы работы с помощью добавления аннотации @EnableAutoConfiguration или составных аннотаций @SpringBootApplication. Что то же самое (по умолчанию — true)
--classpath, -cp	Добавляет элементы пути к классам; удобна в случае наличия сторонних библиотек. Могу порекомендовать вам создать подкаталог lib/ в корневом каталоге своей программы и добавить туда все классы/JAR-файлы
--no-guess-dependencies	Не пытается предугадывать зависимости. Удобно, если в приложении уже используется аннотация @Grab
--no-guess-imports	Не пытается предугадывать, что нужно импортировать. Удобно, если вы уже включили часть импортов в приложение Groovy. Например, эту опцию можно использовать в приложении Java, поскольку все необходимые классы уже импортируются. Более подробно об этом рассказывается в главе 3 (в посвященном автоконфигурации разделе)
-q, --quiet	Останавливает журналирование. Другими словами, в консоль ничего выводиться не будет
-v, --verbose	Выводит в журнал все, что только можно. Полезно для отладки, поскольку показывает диагностическую информацию для кода и все добавленное в программу. См. дополнительную информацию в главе 3
--watch	Отслеживает изменения в заданном файле (файлах). Полезно, чтобы лишний раз не останавливать и не запускать приложение снова

Для запуска приложения Groovy (приведенного в листинге П.1) достаточно просто выполнить команду:

```
$ spring run app.groovy
```

Выполнение этой команды приводит к тому, что веб-приложение запускается и начинает прослушивать по умолчанию на порту 8080, но это можно переопределить с помощью следующей команды:

```
$ spring run app.groovy -- --server.port=8888
```

Эта команда запускает веб-приложение, причем оно прослушивает на порте 8888. Теперь, если нужно добавить какую-либо стороннюю библиотеку и загрузить зависимости, можно просто выполнить:

```
$ spring run -cp lib/mylib.jar app.groovy
```

А для запуска Java-приложения (см. листинг П.2) можно просто выполнить:

```
$ spring run WebApp.java
```

ПРИМЕЧАНИЕ

Остановить приложения можно, нажав Ctrl+C. При запуске Java-приложения важно добавить ключевое слово package. Никакая иерархия каталогов или их создание не нужны. Если не добавить пакет в список просмотра Spring Boot, запустить приложение будет невозможно, поскольку Spring Boot придется просмотреть все возможные зависимости Spring и оно начнет просматривать все использованные зависимости, начиная с корневого каталога каждой из них, так что будьте внимательны!

Для компиляции нескольких файлов можно применять джокерный символ *. Просто выполните следующую команду:

```
$ spring run *.groovy
```

Если по какой-либо причине вам нужно внести изменения в настройки JVM, можете выполнить команду следующего вида:

```
$ JAVA_OPTS=-Xmx2g spring run app.groovy
```

Данная команда увеличивает размер кучи до 2 Гбайт для приложения `app.groovy`.

Команда test

Команда `test`[1] позволяет запускать сценарии Spring Groovy. Ее синтаксис имеет вид:

```
spring test [опции] <файлы> [--] [аргументы]
```

Доступные опции перечислены в табл. П.2.

Таблица П.2. Опции команды spring test

Опция	Описание
--autoconfigure [Boolean]	Добавляет автоконфигурацию преобразований компилятора (по умолчанию — true)
--classpath, -cp	Добавляет элементы пути к классам, что удобно при наличии сторонних библиотек. Могу порекомендовать вам создать подкаталог lib/ в корневом каталоге своей программы и добавить туда все классы/JAR-файлы
--no-guess-dependencies	Не пытается предугадывать зависимости. Удобно, если в приложении уже используется аннотация @Grab
--no-guess-imports	Не пытается предугадывать, что нужно импортировать. Удобно, если вы уже включили часть импортов в приложение Groovy. Например, эту опцию можно использовать в приложении Java, поскольку все необходимые классы уже импортируются. Более подробно об этом рассказывается в главе 3 (в посвященном автоконфигурации разделе)

Чтобы запустить тест, нужен тест, правда? В листингах П.3–П.5 приведены примеры, использующие известные фреймворки JUnit и Spock.

Листинг П.3. test.groovy

```
class MyTest{
        @Test
        void simple() {
                String str= "JUnit works with Spring Boot"
                assertEquals "JUnit works with Spring Boot",str
        }
}
```

[1] Текущая версия Spring Boot CLI (по-видимому, начиная с версии 2.0.0) больше не поддерживает команду test.

В листинге П.3 приведен простейший модульный тест. Никакие операции импорта не нужны, Spring Boot об этом позаботится. Для выполнения его используется команда:

```
$ spring test test.groovy
```

А теперь посмотрите на модульный тест Spock, приведенный в листинге П.4.

Листинг П.4. spock.groovy

```
@Grab('org.spockframework:spock-core:1.0-groovy-2.4')
import spock.lang.Specification
import org.springframework.boot.test.OutputCapture

class SimpleSpockTest extends Specification {

    @org.junit.Rule
    OutputCapture capture = new OutputCapture()

    def "get output and capture it"() {
        when:
        print 'Spring Boot works with Spock'

        then:
        capture.toString() == 'Spring Boot works with Spock'
    }
}
```

В листинге П.4 показано использование фреймворка Spock: расширение класса `Specification` и описание нужных методов. Для применения фреймворка Spock необходимо импортировать нужные зависимости и добавить их в приложение с помощью аннотации `@Grab`, которая включает в приложение зависимость Spock для Groovy. Цель этого раздела состоит лишь в демонстрации использования Spock. Но если вас интересует дополнительная информация по этому вопросу, можете зайти на сайт http://spockframework.org/. Вся документация находится в http://spockframework.github.io/spock/docs/1.0/index.html.

В листинге П.4 также показана одна из новых возможностей Spring Boot: класс `OutputCapture`. Он позволяет захватывать выводимую в `System.out` и `System.err` информацию. Для запуска этого теста выполните ту же команду, что и выше, только поменяйте имя файла:

```
$ spring test spock.groovy
```

Учтите, что Spring Boot не всегда может определить, что используются сторонние библиотеки, так что обязательно применяйте аннотацию `@Grab` и правильные операции `import`.

Взглянем на модульный тест на Java, приведенный в листинге П.5.

Листинг П.5. MyTest.java

```
import org.junit.Rule;
import org.junit.Test;
import org.springframework.boot.test.OutputCapture;

import static org.hamcrest.Matchers.*;
import static org.junit.Assert.*;

public class MyTest {

        @Rule
        public OutputCapture capture = new OutputCapture();

        @Test
        public void stringTest() throws Exception {
                System.out.println("Spring Boot Test works in Java too!");
                assertThat(capture.toString(),
                  containsString("Spring Boot Test works in Java too!"));
        }
}
```

В листинге П.5 приведен написанный на Java модульный тест. Статический оператор `assertThat` относится к классу `org.junit.Assert`. `containsString` — статический метод из класса `org.hamcrest.Matchers`, предназначенный для сравнения с захваченной строкой. В данном модульном тесте также используется класс `OutputCapture`. Для его запуска достаточно выполнить следующую команду:

```
$ spring test MyTest.java
```

Для тестирования веб-приложения `app.groovy` (см. листинг П.1) можете создать код, приведенный в листинге П.6.

Листинг П.6. test.groovy

```
class SimpleWebTest {

    @Test
    void greetingsTest() {
         assertEquals("Spring Boot Rocks", new WebApp().greetings())
    }

}
```

Для его тестирования просто выполните следующую команду:

```
$ spring test app.groovy test.groovy
```

Эта команда использует предыдущий класс (из листинга П.1) — `WebApp` — и вызывает метод `greetings`, возвращающий строку.

Хотя эти примеры очень просты, они демонстрируют, насколько легко создавать и выполнять тесты с помощью интерфейса командной строки. Более продвинутым модульным и интеграционным тестам с помощью всех возможностей Spring Boot посвящена отдельная глава данной книги.

Команда grab

Команда `grab` скачивает все сценарии Spring Groovy и зависимости Java в каталог `./repository`. Ее синтаксис следующий:

```
spring grab [опции] <файлы> [--] [аргументы]
```

Доступные опции перечислены в табл. П.3.

Таблица П.3. Опции команды grab

Опция	Описание
--autoconfigure [Boolean]	Добавляет автоконфигурацию преобразований компилятора (по умолчанию — true)
--classpath, -cp	Добавляет элементы пути к классам, что удобно при наличии сторонних библиотек. Могу порекомендовать вам создать подкаталог lib/ в корневом каталоге своей программы и добавить туда все классы/JAR-файлы
--no-guess-dependencies	Не пытается предугадывать зависимости. Удобно, если в приложении уже используется аннотация @Grab
--no-guess-imports	Не пытается предугадывать, что нужно импортировать. Удобно, если вы уже включили часть импортов в приложение Groovy. Например, эту опцию можно использовать в приложении Java, поскольку все необходимые классы уже импортируются. Более подробно об этом рассказывается в главе 3 (в посвященном автоконфигурации разделе)

Можете воспользоваться любым из вышеприведенных листингов для выполнения команды `grab`. Например, для листинга П.4 можно выполнить:

```
$ spring grab MyTest.java
```

Если вы заглянете в текущий каталог, то увидите, что там появился подкаталог `repository`, содержащий все зависимости. Команда `grab` удобна, когда требуется выполнить приложение Spring Boot, для которого нужны библиотеки,

без интернет-соединения. Команда `grab` также используется для подготовки приложения перед развертыванием его в облаке (эта полезная команда описана в главе 13).

Команда jar

Команда `jar` создает автономный исполняемый JAR-файл на основе Groovy- или Java-сценария. Ее синтаксис следующий:

```
spring jar [опции] <название jar> <файлы>
```

Доступные опции перечислены в табл. П.4.

Таблица П.4. Опции команды jar

Опция	Описание
--autoconfigure [Boolean]	Добавляет автоконфигурацию преобразований компилятора (по умолчанию — true)
--classpath, -cp	Добавляет элементы пути к классам, что удобно при наличии сторонних библиотек. Могу порекомендовать вам создать подкаталог lib/ в корневом каталоге своей программы и добавить туда все классы/JAR-файлы
--exclude	Паттерн для поиска файлов, которые нужно исключить из итогового JAR-файла
--include	Паттерн для поиска файлов, которые нужно включить в итоговый JAR-файл
--no-guess-dependencies	Не пытается предугадывать зависимости. Удобно, если в приложении уже используется аннотация @Grab
--no-guess-imports	Не пытается предугадывать, что нужно импортировать. Удобно, если вы уже включили часть импортов в приложение Groovy. Например, эту опцию можно использовать в приложении Java, поскольку все необходимые классы уже импортируются. Более подробно об этом рассказывается в главе 3 (в посвященном автоконфигурации разделе)

Можете воспользоваться листингом П.1 (`app.groovy`) и выполнить следующую команду:

```
$ spring jar app.jar app.groovy
```

Если вы заглянете в текущий каталог, то увидите, что там появилось два файла — один с названием `app.jar.original`, а второй — `app.jar`. Единственное различие между ними состоит в том, что `app.jar.original` был создан

системой управления зависимостями (Maven) для создания файла `app.jar`. `app.jar` — достаточно большого JAR-файла, который можно выполнить с помощью следующей команды:

```
$ java -jar app.jar
```

В результате выполнения этой команды будет запущено веб-приложение. Команда `jar` обеспечивает переносимость приложения, в том смысле что его можно перенести на другую машину и запустить в любой операционной системе, где установлен интерпретатор Java, не задумываясь о контейнере приложений. Напомню, что Spring Boot встраивает сервер приложений Tomcat в веб-приложения Spring Boot.

Команда war

Очень похожа на предыдущую команду. Команда `war` создает автономный исполняемый WAR-файл на основе Groovy- или Java-сценария. Ее синтаксис следующий:

```
spring war [опции] <название war> <файлы>
```

Доступные опции перечислены в табл. П.5.

Таблица П.5. Опции команды war

Опция	Описание
--autoconfigure [Boolean]	Добавляет автоконфигурацию преобразований компилятора (по умолчанию — true)
--classpath, -cp	Добавляет элементы пути к классам, что удобно при наличии сторонних библиотек. Могу порекомендовать вам создать подкаталог lib/ в корневом каталоге своей программы и добавить туда все классы/JAR-файлы
--exclude	Паттерн для поиска файлов, которые нужно исключить из итогового JAR-файла
--include	Паттерн для поиска файлов, которые нужно включить в итоговый JAR-файл
--no-guess-dependencies	Не пытается предугадывать зависимости. Удобно, если в приложении уже используется аннотация @Grab
--no-guess-imports	Не пытается предугадывать, что нужно импортировать. Удобно, если вы уже включили часть импортов в приложение Groovy. Например, эту опцию можно использовать в приложении Java, поскольку все необходимые классы уже импортируются. Более подробно об этом рассказывается в главе 3 (в посвященном автоконфигурации разделе)

Можете воспользоваться листингом П.1 (`app.groovy`) для выполнения команды `war` следующим образом:

```
$ spring war app.war app.groovy
```

Если вы заглянете в текущий каталог, то увидите, что там появилось два файла — один с названием `app.war.original`, а второй — `app.war`. Последний можно выполнить с помощью следующей команды:

```
$ java -jar app.war
```

При описании предыдущей команды я упоминал слово «переносимость», помните? А как же в случае WAR-файла? Что ж, WAR-файл можно использовать в уже существующих контейнерах приложений, например, tcServer от Pivotal, Tomcat, WebSphere, Jetty и т. п.

ПРИМЕЧАНИЕ

Для создания переносимого исполняемого приложения можно использовать любую из двух предыдущих команд. Единственное отличие состоит в том, что при выполнении команды war создается «допускающий перемещение» WAR-файл, то есть приложение можно запустить автономно или развернуть его в J2EE-совместимом контейнере.

Команда install

Команда `install` очень похожа на команду `grab`; единственное отличие — необходимо указать, какую библиотеку устанавливать (в формате `coordinate`, то есть `groupId:artifactId:version`; аналогично аннотации `@Grab`). После этого команда `install` скачивает библиотеку и зависимости в каталог `lib`. Ее синтаксис следующий:

```
spring install [опции] <координаты>
```

Доступные опции перечислены в табл. П.6.

Возьмем для примера листинг П.4 (`spock.groovy`). Если выполнить следующую команду:

```
$ spring install org.spockframework:spock-core:1.0-groovy-2.4
```

то в каталоге `lib` появятся библиотека Spock и ее зависимости.

Таблица П.6. Опции команды install

Опция	Описание
--autoconfigure [Boolean]	Добавляет автоконфигурацию преобразований компилятора (по умолчанию — true)
--classpath, -cp	Добавляет элементы пути к классам, что удобно при наличии сторонних библиотек. Могу порекомендовать вам создать подкаталог lib/ в корневом каталоге своей программы и добавить туда все классы/JAR-файлы
--no-guess-dependencies	Не пытается предугадывать зависимости. Удобно, если в приложении уже используется аннотация @Grab
--no-guess-imports	Не пытается предугадывать, что нужно импортировать. Удобно, если вы уже включили часть импортов в приложение Groovy. Например, эту опцию можно использовать в приложении Java, поскольку все необходимые классы уже импортируются. Более подробно об этом рассказывается в главе 3 (в посвященном автоконфигурации разделе)

ПРИМЕЧАНИЕ

При использовании утилиты SDKMAN (http://sdkman.io/) библиотеки скачиваются в каталог $HOME/.sdkman/candidates/springboot/1.3.X.RELEASE/lib.

Команда uninstall

Команда `uninstall` удаляет зависимости из каталога `lib`. Ее синтаксис следующий:

```
spring uninstall [опции] <координаты>
```

Доступные опции перечислены в табл. П.7.

Можете попробовать эту команду в действии, выполнив следующее:

```
$ spring uninstall org.spockframework:spock-core:1.0-groovy-2.4
```

Эта команда удаляет все зависимости Spock из каталога `lib`.

Таблица П.7. Опции команды uninstall

Опция	Описание
--autoconfigure [Boolean]	Добавляет автоконфигурацию преобразований компилятора (по умолчанию — true)
--classpath, -cp	Добавляет элементы пути к классам, что удобно при наличии сторонних библиотек. Могу порекомендовать вам создать подкаталог lib/ в корневом каталоге своей программы и добавить туда все классы/JAR-файлы
--no-guess-dependencies	Не пытается предугадывать зависимости. Удобно, если в приложении уже используется аннотация @Grab
--no-guess-imports	Не пытается предугадывать, что нужно импортировать. Удобно, если вы уже включили часть импортов в приложение Groovy. Например, эту опцию можно использовать в приложении Java, поскольку все необходимые классы уже импортируются. Более подробно об этом рассказывается в главе 3 (в посвященном автоконфигурации разделе)

Команда init

Команда `init` помогает в создании новых проектов с помощью Spring Initializr (http://start.spring.io/). Вне зависимости от того, используете ли вы IDE, эта команда поможет подготовить все для разработки приложения Spring Boot. Ее синтаксис следующий:

```
spring init [опции] [местоположение]
```

Доступные опции перечислены в табл. П.8.

Таблица П.8. Опции команды init

Опция	Описание
-a, --artifactId	Координата проекта; по умолчанию (если не указана) — demo
-b, --boot-version	Версия Spring Boot; если не указана, применяется последняя, описанная как parent-pom
--build	Используемая система сборки; возможные значения — maven и gradle. Если не указана, по умолчанию применяется maven

Опция	Описание
-d, --dependencies	Разделенный запятыми список идентификаторов включенных зависимостей. Например, -d=web или -d=web,jdbc,actuator
--description	Описание проекта
-f, --force	Уже существующие файлы перезаписываются
--format	Формат генерируемого содержимого. Удобно для импорта проектов в IDE, например, STS. Возможные значения этой опции — build и project. По умолчанию — project
-g, --groupId	Координаты проекта, ID группы. По умолчанию — com.example
-j, --java-version	Версия языка. По умолчанию — 1.8
-l, --language	Задает язык программирования. Возможные значения — java и groovy. По умолчанию — java
-n, --name	Название приложения. По умолчанию — demo
-p, --packaging	Вариант упаковки проекта. Возможные значения — jar, war и zip. Если не указан, по умолчанию генерирует ZIP-файл
--package-name	Название пакета. По умолчанию — demo
-t, --type	Тип проекта. Возможные значения — maven-project, maven-build, gradle-project и gradle-build. По умолчанию — maven-project
--target	URL используемого сервиса. По умолчанию — https://start.spring.io. Эта опция означает, что вы можете применять свои собственные сервисы
-v, --version	Версия проекта. По умолчанию — 0.0.1-SNAPSHOT
-x, --extract	Извлекает содержимое созданного проекта в текущий каталог, если местоположение не указано

Эту команду вам предстоит использовать очень часто (если только вы не применяете IDE, например, STS или IntelliJ), так что я покажу несколько ее примеров, чтобы вы могли к ней привыкнуть.

Для создания проекта по умолчанию нужно просто выполнить:

```
$ spring init
```

Эта команда генерирует файл `demo.zip`. Можете разархивировать его и посмотреть на структуру (структуру проекта Maven), приведенную на рис. П.1, но самое важное в нем — файл `pom.xml`. Если взглянуть на этот файл, можно увидеть минимальный набор зависимостей: `spring-boot-starter` и `spring-boot-starter-test`.

```
.
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    ├── main
    |   ├── java
    |   |   └── com
    |   |       └── example
    |   |           └── DemoApplication.java
    |   └── resources
    |       └── application.properties
    └── test
        └── java
            └── com
                └── example
                    └── DemoApplicationTests.java

10 directories, 6 files
```

Рис. П.1. Содержимое файла demo.zip

На рис. П.1 показана структура файла `demo.zip`. Обратите внимание на каталог `src`, содержащий файл `main/java/com/example/DemoApplication.java` и, конечно, модульный тест для него. Он также содержит два дополнительных файла, `mvnw` (для UNIX) и `mvnw.cmd` (для Windows). С помощью этих команд вы можете запустить проект Maven, даже если сам Maven в вашей системе не установлен.

Можете просто выполнить следующую команду:

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

Эта команда скачивает (в каталог `.mvn`) и запускает утилиту Maven. Если вы взглянете на файл `DemoApplication.java`, то обнаружите, что он практически ничего не делает, просто запускает приложение Spring Boot. Все это представляет собой шаблон, который можно использовать снова и снова. Для создания веб-приложения достаточно всего лишь добавить зависимость `spring-boot-starter-web`.

Примеры использования команды init

В этом разделе приведены дополнительные примеры использования команды `init`. Следующая команда создает веб-приложение JDBC с типом проекта Gradle:

```
$ spring init -d=web,jdbc --build=gradle
```

Она генерирует файл `demo.zip` с ориентированным на Gradle содержимым. Он включает адаптер Gradle, так что его не нужно будет устанавливать.

Если хотите сгенерировать только файл `pom.xml` (для проекта Maven) или файл `build.gradle` (для проекта Gradle), просто добавьте `--format=build` и `--build=[gradle|maven]`:

```
$ spring init -d=web,data-jpa,security --format=build --build=gradle
```

Эта команда создает файл `build.gradle` с зависимостями Web, JPA и Security:

```
$ spring init -d=jdbc,amqp --format=build
```

Эта команда создает файл `pom.xml`. Если не добавить параметр `--build`, по умолчанию будет создан файл для Maven.

Для создания проекта с названием, `groupId` и `artifactId`, необходимо использовать соответственно параметры `-name`, `-g` и `-a`.

```
$ spring init -d=amqp -g=com.apress.spring -a=spring-boot-rabbitmq
-name=spring-boot-rabbitmq
```

Эта команда создаст файл `spring-boot-rabbitmq.zip` (проект Maven) с заданными `groupId` и `artifactId`.

По умолчанию, если название проекта не указано, используется название `com.example`. Для добавления конкретного названия пакета необходимо добавить параметр `--package-name`.

```
$ spring init -d=web,thymeleaf,data-jpa,data-rest,security -g=com.apress.
spring -a=spring-boot-journal-oauth --package-name=com.apress.spring
-name=spring-boot-journal-oauth
```

ZIP-файл удобен переносимостью, но его можно разархивировать прямо в текущий каталог. Нужно просто добавить параметр `-x`.

```
$ spring init -d=web,thymeleaf,data-jpa,data-rest,security,actuator,h2,mysql
-g=com.apress.spring -a=spring-boot-journal-cloud --package-name=com.apress.
spring -name=spring-boot-journal-cloud -x
```

Эта команда разархивирует ZIP-файл в процессе работы, и его содержимое записывается в текущий каталог.

Если вам интересно узнать больше о зависимостях или других значениях параметров, можете выполнить следующую команду:

```
$ spring init --list
```

Команда `init` используется повсеместно в этой книге, так что не пожалейте времени на изучение всех ее опций.

Альтернатива команде init

Иногда бывает, что нужен только файл `pom.xml` или `build.gradle`, например для проверки зависимостей и объявлений или чтобы взглянуть на объявления плагинов. Для этого можно выполнить следующую команду:

```
$ curl -s https://start.spring.io/pom.xml -d packaging=war -o pom.xml
```

Да, вы все правильно поняли! Как вы помните, команда `init` обращается к сервису Spring Initializr, расположенному по адресу https://start.spring.io, так что можно воспользоваться для этой цели командой `cURL` операционной системы Unix. Данная команда генерирует только файл `pom.xml`. А если вам интересно, что еще можно сделать с помощью команды `cURL` операционной системы Unix, просто выполните следующее[1]:

```
$ curl start.spring.io
```

Эта команда выводит в терминал все имеющиеся опции и несколько примеров использования `cURL` со Spring Initializr. Как вы знаете из предыдущих глав, в IDE STS (Spring Tool Suite) можно создать приложение Spring Boot, выбрав мастер Spring Starter Project. Этот мастер подключается к Spring Initializr, так что для получения структуры проекта Spring Boot можно использовать либо IDE, либо командную строку.

Команда shell

Команда `shell` запускает встроенную командную оболочку. Выполните следующую команду:

```
$ spring shell
Spring Boot (v1.3.X.RELEASE)
Hit TAB to complete. Type 'help' and hit RETURN for help, and 'exit' to quit.
$
```

[1] В таком виде команда не работает; только с указанием протокола HTTPS:

$ curl https://start.spring.io

Как вы видите из выведенного в консоль, можно ввести help для получения дополнительной информации об этой командной оболочке. Фактически spring shell включает описанные выше команды, но выполняемые во встроенной командной оболочке. Одно из ее преимуществ — наличие автодополнения табуляцией, что позволяет получить все возможные варианты опций.

Команда help

Команда help — ваш главный помощник. Вот результаты ее выполнения:

```
spring help
usage: spring [--help] [--version]
       <команда> [<аргументы>]

Available commands are.

  run [options] <files> [--] [args]
    Run a spring groovy script

  test [options] <files> [--] [args]
    Run a spring groovy script test

  grab
    Download a spring groovy script's dependencies to ./repository

  jar [options] <jar-name> <files>
    Create a self-contained executable jar file from a Spring Groovy script

  war [options] <war-name> <files>
    Create a self-contained executable war file from a Spring Groovy script

  install [options] <coordinates>
    Install dependencies to the lib directory

  uninstall [options] <coordinates>
    Uninstall dependencies from the lib directory

  init [options] [location]
    Initialize a new project using Spring Initializr (start.spring.io)

  shell
    Start a nested shell

Common options.
  -d, --debug Verbose mode
    Print additional status information for the command you are running

See 'spring help <command>' for more information on a specific command.
```

Как видите из этой выведенной в консоль информации, можно также выполнить команду `spring help <команда>`, что очень удобно, поскольку она выводит дополнительную информацию о нужной команде и несколько примеров ее использования. Например, если хотите узнать больше о команде `init`, просто выполните следующее:

```
$ spring help init
```

Помните, команда `spring help` — ваш главный помощник.

Резюме

Из этого приложения вы узнали, как использовать интерфейс командной строки Spring Boot. В нем описываются все доступные команды и их опции.

Вы узнали, помимо прочего, что одной из важнейших команд является `init`, которая используется во всей этой книге либо через терминал в командной строке, либо с помощью IDE, например, STS или IntelliJ.

Фелипе Гутьеррес

Spring Boot 2: лучшие практики для профессионалов

Перевел с английского *И. Пальти*

Завсдующая рсдакцисй	*Ю. Сергиенко*
Руководитсль проскта	*С. Давид*
Всдущий рсдактор	*Н. Гринчик*
Литсратурный рсдактор	*Е. Рафалюк-Бузовская*
Художсствснный рсдактор	*В. Мостипан*
Коррсктор	*О. Андриевич, Е. Павлович*
Всрстка	*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: 06.2020. 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

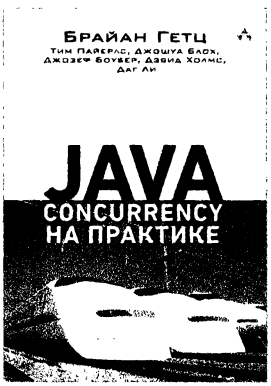
Подписано в печать 20.05.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 37,410. Тираж 700. Заказ 3049.

Отпечатано в АО «Первая Образцовая типография» Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул Полиграфистов, 1
Сайт www chpd ru, E-mail sales@chpd ru
тел 8(499) 270-73-59

Рауль-Габриэль Урма, Марио Фуско, Алан Майкрофт

СОВРЕМЕННЫЙ ЯЗЫК JAVA. ЛЯМБДА-ВЫРАЖЕНИЯ, ПОТОКИ И ФУНКЦИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Преимущество современных приложений — в передовых решениях, включающих микросервисы, реактивные архитектуры и потоковую обработку данных. Лямбда-выражения, потоки данных и долгожданная система модулей платформы Java значительно упрощают их реализацию. Пришло время повысить свою квалификацию и встретить любой вызов во всеоружии! Книга поможет вам овладеть новыми возможностями современных дополнений, таких как API Streams и система модулей платформы Java. Откройте для себя новые подходы к конкурентности и узнайте, как концепции функциональности улучшают работу с кодом.



Брайан Гетц, Тим Пайерлс, Джошуа Блох, Джозеф Боубер

JAVA CONCURRENCY НА ПРАКТИКЕ

Потоки являются фундаментальной частью платформы Java. Многоядерные процессоры — это обыденная реальность, а эффективное использование параллелизма стало необходимым для создания любого высокопроизводительного приложения. Улучшенная виртуальная машина Java, поддержка высокопроизводительных классов и богатый набор строительных блоков для задач распараллеливания стали в свое время прорывом в разработке параллельных приложений. В «Java Concurrency на практике» сами создатели прорывной технологии объясняют не только принципы работы, но и рассказывают о паттернах проектирования.

Аллен Б. Доуни

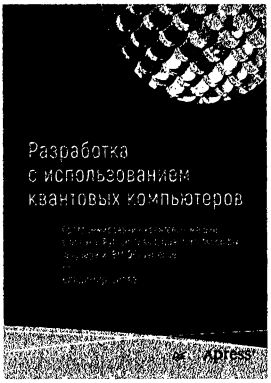
АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ. ИЗВЛЕЧЕНИЕ ИНФОРМАЦИИ НА ЯЗЫКЕ JAVA

Изучите, как следует реализовывать эффективные алгоритмы на основе важнейших структур данных на языке Java, а также как измерять производительность этих алгоритмов. Каждая глава сопровождается упражнениями, помогающими закрепить материал.

Б. Эккель

ФИЛОСОФИЯ JAVA. 4-Е ПОЛНОЕ ИЗД.

Впервые читатель может познакомиться с полной версией этого классического труда, который ранее на русском языке печатался в сокращении. Книга, выдержавшая в оригинале не одно переиздание, за глубокое и поистине философское изложение тонкостей языка Java считается одним из лучших пособий для программистов. Чтобы по-настоящему понять язык Java, необходимо рассматривать его не просто как набор неких команд и операторов, а понять его «философию», подход к решению задач, в сравнении с таковыми в других языках программирования. На этих страницах автор рассказывает об основных проблемах написания кода: в чем их природа и какой подход использует Java в их разрешении. Поэтому обсуждаемые в каждой главе черты языка неразрывно связаны с тем, как они используются для решения определенных задач.



Владимир Силва

РАЗРАБОТКА С ИСПОЛЬЗОВАНИЕМ КВАНТОВЫХ КОМПЬЮТЕРОВ

Квантовые вычисления не просто меняют реальность! Совершенно новая отрасль рождается на наших глазах, чтобы создать немыслимое ранее и обесценить некоторые достижения прошлого. В этой книге рассмотрены наиболее важные компоненты квантового компьютера: кубиты, логические вентили и квантовые схемы, а также объясняется отличие квантовой архитектуры от традиционной. Вы сможете бесплатно экспериментировать с ними как в симуляторе, так и на реальном квантовом устройстве с применением IBM Q Experience. Вы узнаете, как выполняются квантовые вычисления с помощью QISKit (программный инструментарий для обработки квантовой информации), Python SDK и других API, в частности QASM.

Мехди Меджуи, Эрик Уайлд, Ронни Митра, Майк Амундсен

НЕПРЕРЫВНОЕ РАЗВИТИЕ API. ПРАВИЛЬНЫЕ РЕШЕНИЯ В ИЗМЕНЧИВОМ ТЕХНОЛОГИЧЕСКОМ ЛАНДШАФТЕ

Для реализации API необходимо провести большую работу. Чрезмерное планирование может стать пустой тратой сил, а его недостаток приводит к катастрофическим последствиям. В этой книге вы получите решения, которые позволят вам распределить необходимые ресурсы и достичь требуемого уровня эффективности за оптимальное время. Как соблюсти баланс гибкости и производительности, сохранив надежность и простоту настройки? Четыре эксперта из Академии API объясняют разработчикам ПО, руководителям продуктов и проектов, как максимально увеличить ценность их API, управляя интерфейсами как продуктами с непрерывным жизненным циклом.

БИБЛИОТЕКА ПРОГРАММИСТА

Хотите повысить свою эффективность в разработке корпоративных и облачных Java-приложений?

Увеличьте скорость и простоту разработки микросервисов и сложных приложений, избавившись от забот по конфигурации Spring.

Используйте Spring Boot 2 и такие инструменты фреймворка Spring 5, как WebFlux, Security, Actuator, а также фреймворк Micrometer, предоставляющий новый способ сбора метрик.

- *Настройка и использование Spring Boot 2.*
- *Работа с нефункциональными требованиями с помощью Spring Boot Actuator.*
- *Разработка веб-приложений с помощью Spring Boot 2.*
- *Постоянное хранение данных с использованием JDBC, JPA и баз данных NoSQL.*
- *Обмен сообщениями с помощью JMS, RabbitMQ и WebSockets.*
- *Краткое знакомство с проектами Spring Cloud.*
- *Тестирование, микросервисы и развертывание в облачной среде с использованием Spring Boot 2.*
- *Расширение Spring Boot путем создания своего собственного «стартового набора» и добавления функциональности @Enable.*

Фелипе Гутьеррес — ведущий архитектор компании Pivotal, разработчик Spring Boot. За плечами Фелипе более 20 лет опыта работы в сфере IT. Он принимал участие в разработке программ для правительства, систем здравоохранения и образования, сферы банковских услуг, компаний по розничной торговле. Был консультантом таких крупных компаний, как Nokia, Apple, Redbox и Qualcomm.

ПИТЕР®

Заказ книг:
тел.: (812) 703-73-74
books@piter.com

WWW.PITER.COM
каталог книг и интернет-магазин

instagram.com/piterbooks
youtube.com/ThePiterBooks
vk.com/piterbooks
facebook.com/piterbooks

ISBN:978-5-4461-1587-7