

Spring

в действии

ШЕСТОЕ ИЗДАНИЕ

Крейг Уолс



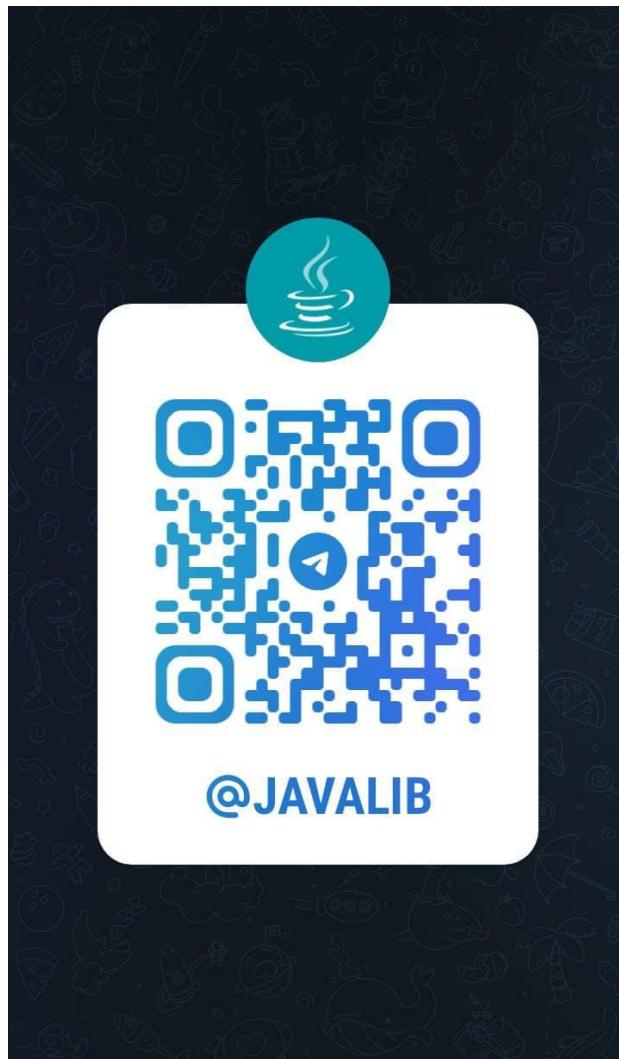
MANNING



Крейг Уоллс

Spring в действии

Ещё больше книг по Java в нашем телеграм канале:
<https://t.me/javalib>



Spring in Action

Sixth Edition

CRAIG WALLS



MANNING
Shelter Island

Spring в действии

6-е издание

КРЕЙГ УОЛЛС



Москва, 2022

УДК 004.432

ББК 32.972.1

У63

Уоллс К.

У63 Spring в действии. 6-е изд. / пер. с англ. А. Н. Киселева. – М.: ДМК Пресс, 2022. – 544 с.: ил.

ISBN 978-5-93700-112-2

Это исчерпывающее руководство по основным возможностям Spring, написанное простым и ясным языком. Книга шаг за шагом проведет вас по пути создания законченного веб-приложения на основе базы данных. Новое 6-е издание охватывает не только основы Spring, но также новые возможности, такие как реактивные потоки или интеграция с Kubernetes и RSocket. Независимо от того, впервые ли вы знакомитесь с фреймворком Spring или переходите на новую версию Spring 5.3, этот классический бестселлер станет вашей настольной книгой.

Издание предназначено Java-разработчикам, как уже использующим, так и только начинающим применять в своей работе Spring.

УДК 004.432

ББК 32.972.1

Original English language edition published by Manning Publications USA. Copyright © 2022 by Manning Publications. Russian-language edition copyright © 2022 by DMK Press. All rights reserved.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Оглавление

Часть I ■ ОСНОВЫ SPRING	24
1 ■ Введение в Spring	25
2 ■ Разработка веб-приложений	54
3 ■ Работа с данными	88
4 ■ Работа с нереляционными данными	123
5 ■ Безопасность в Spring.....	144
6 ■ Работа с конфигурацией	173
Часть II ■ ИНТЕГРАЦИЯ С ПРИЛОЖЕНИЯМИ SPRING	196
7 ■ Создание служб REST	197
8 ■ Безопасность REST API.....	222
9 ■ Асинхронная передача сообщений.....	247
10 ■ Интеграция Spring	283
Часть III ■ РЕАКТИВНЫЙ SPRING	317
11 ■ Введение в Reactor.....	318
12 ■ Разработка реактивных API.....	350
13 ■ Реактивное хранение данных.....	382
14 ■ RSocket.....	416
Часть IV ■ РАЗВЕРТЫВАНИЕ SPRING	433
15 ■ Spring Boot Actuator	434
16 ■ Администрирование Spring	472
17 ■ Мониторинг Spring с помощью JMX.....	484
18 ■ Развёртывание Spring	492

Содержание

<i>Из отзывов к пятому изданию книги Крейга Уоллса «Spring в действии»</i>	13
<i>Предисловие</i>	14
<i>Благодарности</i>	15
<i>Об этой книге</i>	17
<i>Об авторе</i>	22
<i>Об иллюстрации на обложке</i>	23

Часть I ОСНОВЫ SPRING 24

1 Введение в Spring	25
1.1 Что такое Spring?	26
1.2 Инициализация приложения Spring	29
1.2.1 Инициализация проекта Spring с помощью Spring Tool Suite	30
1.2.2 Структура проекта Spring	34
1.3 Разработка приложения Spring	41
1.3.1 Обработка веб-запросов	41
1.3.2 Определение представления	42
1.3.3 Тестирование контроллера	43
1.3.4 Сборка и запуск приложения	45
1.3.5 Spring Boot DevTools	47
1.3.6 Обзор результатов	49
1.4 Обзор ландшафта Spring	50
1.4.1 Ядро Spring Framework	50
1.4.2 Spring Boot	51
1.4.3 Spring Data	51
1.4.4 Spring Security	52
1.4.5 Spring Integration и Spring Batch	52
1.4.6 Spring Cloud	52
1.4.7 Spring Native	53
Итоги	53

2 Разработка веб-приложений	54
2.1 Отображение информации	55
2.1.1 Предметная область ингредиентов	56

2.1.2	<i>Создание класса контроллера</i>	60
2.1.3	<i>Создание представления</i>	63
2.2	<i>Обработка отправленной формы</i>	68
2.3	<i>Проверка данных в форме</i>	75
2.3.1	<i>Объявление правил проверки</i>	76
2.3.2	<i>Выполнение проверки после привязки</i>	79
2.3.3	<i>Отображение сообщений об ошибках</i>	80
2.4	<i>Работа с контроллерами представлений</i>	82
2.5	<i>Выбор механизма шаблонов для создания представлений</i>	84
2.5.1	<i>Кеширование шаблонов</i>	85
	<i>Итоги</i>	86
3	Работа с данными	88
3.1	<i>Чтение и запись данных с помощью JDBC</i>	89
3.1.1	<i>Подготовка объектов данных к хранению</i>	91
3.1.2	<i>Использование JdbcTemplate</i>	92
3.1.3	<i>Определение схемы и предварительная загрузка данных</i>	98
3.1.4	<i>Сохранение данных</i>	101
3.2	<i>Spring Data JDBC</i>	106
3.2.1	<i>Добавление Spring Data JDBC в спецификацию сборки</i>	107
3.2.2	<i>Определение интерфейсов репозиториев</i>	107
3.2.3	<i>Аннотирование классов данных предметной области</i>	109
3.2.4	<i>Предварительная загрузка данных с помощью CommandLineRunner</i>	112
3.3	<i>Хранение данных с помощью Spring Data JPA</i>	114
3.3.1	<i>Добавление зависимости Spring Data JPA</i>	114
3.3.2	<i>Аннотирование классов данных предметной области</i>	115
3.3.3	<i>Объявление репозиториев JPA</i>	119
3.3.4	<i>Специализация репозиториев</i>	119
	<i>Итоги</i>	122
4	Работа с нереляционными данными	123
4.1	<i>Репозитории в Cassandra</i>	124
4.1.1	<i>Включение Spring Data Cassandra</i>	124
4.1.2	<i>Моделирование данных в Cassandra</i>	128
4.1.3	<i>Отображение типов данных предметной области для хранения в Cassandra</i>	129
4.1.4	<i>Определение репозиториев Cassandra</i>	135
4.2	<i>Определение репозиториев MongoDB</i>	136
4.2.1	<i>Включение Spring Data MongoDB</i>	137
4.2.2	<i>Отображение типов данных предметной области в документы</i>	138
4.2.3	<i>Определение репозиториев MongoDB</i>	141
	<i>Итоги</i>	143
5	Безопасность в Spring	144
5.1	<i>Включение Spring Security</i>	145
5.2	<i>Настройка аутентификации</i>	147
5.2.1	<i>Служба хранения сведений о пользователях в памяти</i>	149
5.2.2	<i>Настройка аутентификации пользователя</i>	150

5.3	Защита веб-запросов	157
5.3.1	Защита запросов	157
5.3.2	Создание страницы входа	160
5.3.3	Использование сторонних систем аутентификации	163
5.3.4	Предотвращение подделки межсайтовых запросов	166
5.4	Безопасность на уровне методов	167
5.5	Знай своего пользователя	169
	Итоги	172

6 Работа с конфигурацией.....173

6.1	Тонкая настройка автоконфигурации	174
6.1.1	Абстракция окружения Spring.....	175
6.1.2	Настройка источника данных	176
6.1.3	Настройка встроенного сервера	178
6.1.4	Настройка журналирования	179
6.1.5	Использование специальных значений свойств	181
6.2	Создание своих конфигурационных свойств	182
6.2.1	Определение хранителей конфигурационных свойств	184
6.2.2	Объявление метаданных конфигурационного свойства	187
6.3	Настройка с помощью профилей	190
6.3.1	Определение свойств профиля	191
6.3.2	Активация профилей	192
6.3.3	Условное создание bean-компонентов для профилей.....	193
	Итоги	195

Часть II ИНТЕГРАЦИЯ С ПРИЛОЖЕНИЯМИ SPRING196

7 Создание служб REST.....197

7.1	Создание контроллеров RESTful	198
7.1.1	Извлечение данных с сервера	198
7.1.2	Отправка данных на сервер	204
7.1.3	Изменение данных на сервере.....	205
7.1.4	Удаление данных на сервере	208
7.2	Включение услуг на основе данных	209
7.2.1	Настройка имен отношений и путей к ресурсам	212
7.2.2	Деление на страницы и упорядочение.....	214
7.3	Использование служб REST	215
7.3.1	Получение ресурса запросом GET.....	217
7.3.2	Отправка ресурса запросом PUT.....	219
7.3.3	Удаление ресурса запросом DELETE.....	219
7.3.4	Отправка ресурса запросом POST.....	220
	Итоги	221

8 Безопасность REST API.....222

8.1	Знакомство с OAuth 2	223
8.2	Создание сервера авторизации	229

8.3	Защита API с помощью сервера ресурсов.....	238
8.4	Разработка клиента	241
Итоги		246

9 Асинхронная передача сообщений.....247

9.1	Отправка сообщений с помощью JMS	248
9.1.1	Настройка JMS	249
9.1.2	Отправка сообщений с помощью <i>JmsTemplate</i>	251
9.1.3	Получение сообщений с помощью JMS	260
9.2	RabbitMQ и AMQP	264
9.2.1	Добавление поддержки RabbitMQ в приложение Spring	266
9.2.2	Отправка сообщений с помощью <i>RabbitTemplate</i>	267
9.2.3	Получение сообщений из RabbitMQ	271
9.3	Обмен сообщениями с помощью Kafka	276
9.3.1	Настройка Spring для обмена сообщениями через Kafka	277
9.3.2	Отправка сообщений с помощью <i>KafkaTemplate</i>	278
9.3.3	Получение сообщений из Kafka.....	280
Итоги		282

10 Интеграция Spring.....283

10.1	Обявление простого потока интеграции	284
10.1.1	Определение потоков интеграции в XML	286
10.1.2	Определение потоков интеграции в коде на Java	288
10.1.3	Конфигурация на <i>Spring Integration DSL</i>	290
10.2	Обзор ландшафта Spring Integration	291
10.2.1	Каналы сообщений	292
10.2.2	Фильтры	294
10.2.3	Преобразователи.....	295
10.2.4	Маршрутизаторы	296
10.2.5	Сплиттеры	298
10.2.6	Активаторы служб	301
10.2.7	Шлюзы	303
10.2.8	Адаптеры каналов	304
10.2.9	Модули конечных точек	306
10.3	Создание потока интеграции для электронной почты	308
Итоги		316

Часть III РЕАКТИВНЫЙ SPRING.....317

11 Введение в Reactor.....318

11.1	Основы реактивного программирования	319
11.1.1	Определение реактивных потоков данных	321
11.2	Reactor	323
11.2.1	Диаграммы реактивных потоков	325
11.2.2	Добавление зависимости от Reactor	326
11.3	Использование распространенных реактивных операций.....	327
11.3.1	Создание реактивных типов.....	327
11.3.2	Комбинирование реактивных типов	332

11.3.3	Преобразование и фильтрация реактивных потоков	336
11.3.4	Выполнение логических операций с реактивными типами.....	347
Итоги	349

12 Разработка реактивных API 350

12.1	Spring WebFlux	350
12.1.1	Введение в Spring WebFlux	352
12.1.2	Создание реактивных контроллеров.....	354
12.2	Определение обработчиков запросов в функциональном стиле	359
12.3	Тестирование реактивных контроллеров.....	363
12.3.1	Тестирование запросов GET.....	363
12.3.2	Тестирование запросов POST.....	366
12.3.3	Тестирование с использованием действующего сервера	367
12.4	Реактивный REST API.....	368
12.4.1	Получение ресурсов	369
12.4.2	Отправка ресурсов	371
12.4.3	Удаление ресурсов	372
12.4.4	Обработка ошибок	373
12.4.5	Обмен запросами	375
12.5	Защита реактивного веб-API	376
12.5.1	Реактивная модель настройки безопасности	377
12.5.2	Настройка реактивной службы учетных записей	379
Итоги	380

13 Реактивное хранение данных 382

13.1	R2DBC	383
13.1.1	Определение сущностей предметной области для R2DBC	384
13.1.2	Определение реактивных репозиториев	389
13.1.3	Тестирование репозиториев R2DBC	391
13.1.4	Определение службы управления агрегатами в OrderRepository	393
13.2	Реактивное хранилище документов в MongoDB.....	399
13.2.1	Определение типов документов	400
13.2.2	Определение реактивных репозиториев MongoDB	403
13.2.3	Тестирование реактивных репозиториев MongoDB	404
13.3	Реактивное хранилище данных в Cassandra	407
13.3.1	Определение классов предметной области для Cassandra	408
13.3.2	Создание реактивных репозиториев Cassandra.....	412
13.3.3	Тестирование реактивных репозиториев Cassandra	413
Итоги	415

14 RSocket 416

14.1	Введение в RSocket	417
14.2	Создание простого сервера и клиента RSocket	419
14.2.1	Реализация модели «запрос-ответ»	420
14.2.2	Реализация модели «запрос-поток».....	423
14.2.3	Реализация модели «запустил и забыл»	425
14.2.4	Двунаправленная передача сообщений	427
14.3	Передача по протоколу RSocket через WebSocket	431
Итоги	432

Часть IV РАЗВЕРТЫВАНИЕ SPRING	433
15 Spring Boot Actuator	434
15.1 Введение в Actuator	435
15.1.1 Настройка базового пути Actuator	436
15.1.2 Включение и отключение конечных точек Actuator	436
15.2 Использование конечных точек Actuator	438
15.2.1 Получение важной информации о приложении	439
15.2.2 Просмотр сведений о конфигурации	442
15.2.3 Наблюдение за действиями приложения	451
15.2.4 Получение метрик времени выполнения	453
15.3 Настройка Actuator	457
15.3.1 Добавление информации в конечную точку /info	457
15.3.2 Определение своих индикаторов работоспособности	462
15.3.3 Регистрация пользовательских метрик	464
15.3.4 Создание пользовательских конечных точек	466
15.4 Защита конечных точек Actuator	469
Итоги	471
16 Администрирование Spring	472
16.1 Spring Boot Admin	473
16.1.1 Создание сервера Admin	473
16.1.2 Регистрация клиентов сервера Admin	475
16.2 Исследование возможностей сервера Admin	476
16.2.1 Обзор общего состояния приложения	477
16.2.2 Просмотр ключевых метрик	478
16.2.3 Исследование свойств окружения	478
16.2.4 Просмотр и изменение уровней журналирования	480
16.3 Защита сервера Admin	481
16.3.1 Включение регистрации на сервере Admin	481
16.3.2 Аутентификация в Actuator	482
Итоги	483
17 Мониторинг Spring с помощью JMX	484
17.1 Работа с компонентами MBean в Actuator	484
17.2 Создание своих компонентов MBean	487
17.3 Отправка уведомлений	489
Итоги	491
18 Развертывание Spring	492
18.1 Варианты развертывания	493
18.2 Сборка выполняемых файлов JAR	494
18.3 Сборка образа контейнера	495
18.3.1 Развертывание в Kubernetes	499
18.3.2 Корректное завершение работы	501
18.3.3 Проверка готовности и жизнеспособности приложения	502
18.4 Создание и развертывание файлов WAR	506
18.5 Закончим тем, с чего начинали	508
Итоги	508

A	Создание проектов приложений Spring	509
A.1	Инициализация проекта с помощью Spring Tool Suite.....	509
A.2	Инициализация проекта с помощью IntelliJ IDEA.....	513
A.3	Инициализация проекта с помощью NetBeans.....	514
A.4	Инициализация проекта с помощью start.spring.io.....	519
A.5	Инициализация проекта из командной строки	522
A.6	Сборка и запуск проектов	525
	<i>Предметный указатель.....</i>	527

Ещё больше книг по Java в нашем телеграм канале:
<https://t.me/javalib>



Из отзывов к пятому изданию книги Крейга Уоллса «Spring в действии»

«Отличный источник знаний о таком сложном фреймворке».

– *Арнальдо Габриэль Аяяла Мейер (Arnaldo Gabriel Ayala Meyer), Consultores Informáticos S.R.L.*

«Охватывает все аспекты последней версии Spring и демонстрирует их на практических примерах».

– *Билл Флай (Bill Fly), Брукхейвенский колледж*

«Настольная книга для изучения Spring Framework и отличное справочное руководство».

– *Колин Джойс (Colin Joyce), Cisco*

«На мой взгляд, это лучшая книга о Spring. Новое издание подверглось обширному обновлению, обеспечившему уникальный баланс между теорией и практикой. Она поможет вам быстро приступить к работе и даст подробные пояснения».

– *Дэниел Вон (Daniel Vaughan), Европейский институт биоинформатики*

«Искрывающее руководство по разработке облачных приложений с использованием Spring».

– *Дэвид Уизерспун (David Witherspoon), Parsons Corporation*

«Источник истины в последней инстанции для экосистемы Spring».

– *Эдду Мелендес Гонсалес (Eddú Meléndez Gonzales), Scotiabank*

«Я настоятельно рекомендую эту книгу и новичкам в Spring Framework, и опытным разработчикам, которые хотят глубже освоить новейшие возможности, доступные в экосистеме Spring 5».

– *Иэн Кэмпбелл (Iain Campbell), Tango Telecom*

«Даже будучи опытным разработчиком на Spring, я нашел в этой книге много новых практических советов».

– *Джетро Коэнради (Jettro Coenradie), Luminis*

Предисловие

Фреймворк Spring появился более 18 лет назад, и его главной целью было упрощение разработки Java-приложений. Первоначальная цель состояла в том, чтобы дать облегченную альтернативу EJB 2.x. Но это было только начало. С годами фреймворк Spring охватывал все более широкий круг задач разработки, включая хранение данных, безопасность, интеграцию, облачные вычисления и т. д.

Несмотря на то что возраст Spring приближается к двум десятилетиям и он все шире охватывает сферу разработки корпоративных приложений на Java, в развитии фреймворка не наблюдается никаких признаков замедления. Spring продолжает решать проблемы разработки на Java, будь то приложение, развернутое на обычном сервере приложений, или контейнерное приложение, развернутое в кластере Kubernetes в облаке. А учитывая наличие Spring Boot – фреймворка, обеспечивающего автоматическую конфигурацию, помочь в сборке зависимостей и мониторинг времени выполнения, – никогда не было более удачного времени, чтобы стать разработчиком на Spring!

Это издание книги «Spring в действии» – ваш путеводитель по Spring и Spring Boot. Оно было обновлено и теперь еще полнее отражает все, что могут предложить оба фреймворка. Даже если вы только приступаете к изучению Spring, вы сможете запустить свое первое приложение на Spring еще до конца первой главы. По мере изучения книги вы узнаете, как создавать веб-приложения, работать с данными, защищать приложения и управлять их конфигурациями. Затем вы познакомитесь с вариантами интеграции приложений Spring с другими приложениями и узнаете, как извлечь выгоду из реактивного программирования, используя новый протокол RSocket. Ближе к концу вы увидите, как подготовить приложение к передаче в промышленное окружение, и узнаете о доступных вариантах развертывания.

Кем бы вы ни были – новичком, только начинающим изучать Spring, или ветераном, с многолетним опытом разработки на Spring, – эта книга станет вашим следующим шагом в вашем путешествии. Я очень рад представить вам это руководство и с нетерпением жду появления ваших творений на Spring!

Благодарности

Одна из самых замечательных особенностей Spring и Spring Boot – автоматическое создание всей базовой инфраструктуры приложения, что позволяет вам как разработчику сосредоточиться на логике, уникальной для вашего приложения. К сожалению, при написании книг такое невозможно. Или я что-то упустил?

Многие сотрудники в издательстве Manning приложили все силы, чтобы эта книга стала как можно лучше. Большое спасибо моему редактору-консультанту Дженни Старт (Jenny Stout) и выпускающему редактору Дейдре Хайам (Deirdre Hiam), литературному редактору Памеле Хант (Pamela Hunt), художнику Дженифер Хоул (Jennifer Houle) и всей производственной команде за прекрасную работу над этой книгой.

В создании книги участвовало несколько рецензентов, которые вычитывали рукопись и своими бесценными отзывами помогли не сбиться с курса и продолжать освещать то, что действительно важно. За это я благодарю Эла Пезевски (Al Pezewski), Александро Кампейса (Alessandro Campeis), Бекки Хьюэтт (Becky Huett), Кристиана Кройцер-Бека (Christian Kreutzer-Beck), Конора Редмонда (Conor Redmond), Дэвида Паккуда (David Paccoud), Дэвида Торрубиа Иньиго (David Torrubia Iñigo), Дэвида Уизерспуна (David Witherspoon), Германа Гонсалеса-Морриса (German Gonzalez-Morris), Иэна Кэмбелла (Iain Campbell), Джона Гюнтера (Jon Guenther), Кевина Ляо (Kevin Liao), Марка Дешампса (Mark Dechamps), Майкла Брайта (Michael Bright), Филиппа Виалатта (Philippe Vialatte), Пьера-Мишеля Анселя (Pierre-Michel Ansel), Тони Свитса (Tony Sweets), Уильяма Фляя (William Fly) и Зородзайи Мукуя (Zorodzayi Mukuuya).

Я обязательно должен поблагодарить всю команду инженеров, занимающихся разработкой Spring. Вы работаете над развитием одного из самых невероятных фреймворков, которые я когда-либо видел, и я рад считать себя вашим коллегой.

Большое спасибо моим соратникам по конференции «No Fluff/Just Stuff». Я продолжаю учиться у каждого из вас. И большое спасибо всем, кто посетил одну из моих сессий в рамках конференции NFJS;

несмотря на то что я стою за трибуной, я часто многому учусь у вас.

Как и в предыдущем издании, я хотел бы поблагодарить финикийцев. Вы знаете, что вы сделали.

Наконец, спасибо моей прекрасной жене Рейми (Raymie). Ты – любовь всей моей жизни и моя самая сладкая мечта: спасибо за твою поддержку и за то, что выдержала еще один книжный проект. Спасибо моим милым и замечательным девочкам, Мейси (Maisy) и Мади (Madi): я так горжусь вами, и я рад видеть, какими замечательными девушками вы становитесь. Я люблю всех вас больше, чем можно себе представить или выразить словами.

Об этой книге

«Spring в действии, шестое издание» писалась с простой целью: дать вам возможность создавать потрясающие приложения с использованием Spring Framework, Spring Boot и других инструментов из экосистемы Spring. Книга начинается демонстрацией создания веб-приложения на Java с поддержкой базы данных с помощью Spring и Spring Boot. Затем она описывает дополнительные возможности, показывая, как организовать интеграцию с другими приложениями и программами, использующими реактивные типы. Наконец, в ней обсуждаются вопросы подготовки приложения к развертыванию.

Все проекты в экосистеме Spring снабжаются превосходной документацией, однако эта книга дает то, чего не может дать ни один из справочных документов, – практическое руководство по объединению компонентов Spring и созданию действующих приложений.

Кому адресована эта книга

Книга «Spring в действии, шестое издание» адресована всем Java-разработчикам, желающим начать использовать Spring Boot и Spring Framework, а также опытным разработчикам на Spring, которые стремятся выйти за рамки основ и изучить новейшие возможности Spring.

Структура книги

Книга состоит из четырех частей и 18 глав. Часть I охватывает основы создания приложений на Spring:

- глава 1 знакомит с фреймворками Spring и Spring Boot, а также с приемами инициализации проекта на Spring. В этой главе вы сделаете первые шаги к созданию приложения Spring, которое будете развивать на протяжении всей книги;
- глава 2 обсуждает создание веб-уровня приложения с использованием Spring MVC. Здесь вы создадите контроллеры, обрабатывающие веб-запросы, и представления, отображающие информацию в веб-браузере;

- глава 3 рассматривает серверную часть приложения Spring, обеспечивающую хранение информации в реляционной базе данных;
- глава 4 продолжает тему хранения данных и описывает приемы хранения данных в нереляционных базах данных, таких как Cassandra и MongoDB;
- глава 5 демонстрирует приемы использования Spring Security для аутентификации пользователей и предотвращения несанкционированного доступа к приложению;
- глава 6 показывает, как настроить приложение Spring с помощью Spring Boot и как выборочно применять конфигурации с помощью профилей.

Часть II охватывает темы, связанные с интеграцией приложений Spring с другими приложениями:

- глава 7 продолжает обсуждение Spring MVC, начатое в главе 2, и рассматривает вопросы создания и использования REST API в Spring;
- глава 8 показывает, как защитить API, созданные в главе 7, с помощью Spring Security и OAuth 2;
- глава 9 рассматривает приемы асинхронных взаимодействий, позволяющие приложению Spring отправлять и получать сообщения с помощью службы сообщений RabbitMQ или Kafka;
- глава 10 обсуждает декларативную интеграцию приложений с использованием Spring Integration.

Часть III исследует новую захватывающую поддержку реактивного программирования в Spring:

- глава 11 знакомит с Project Reactor – библиотекой реактивного программирования, лежащей в основе реактивных возможностей Spring 5;
- глава 12 вновь рассматривает разработку REST API и представляет Spring WebFlux – новый веб-фреймворк, который многое заимствовал из Spring MVC и предлагает новую реактивную модель для веб-разработки;
- глава 13 разбирает приемы хранения реактивных данных с помощью Spring Data в базах данных Cassandra и Mongo;
- глава 14 знакомит с RSocket, новым коммуникационным протоколом – реактивной альтернативой HTTP для создания API.

В части IV рассказывается, как подготовить и развернуть приложение в промышленном окружении:

- глава 15 знакомит с Spring Boot Actuator – расширением Spring Boot, помогающим экспортовать функции приложения Spring в виде конечных точек REST;
- глава 16 показывает, как использовать Spring Boot Admin для создания удобного браузерного приложения администрирования поверх Actuator;

- глава 17 обсуждает отображение и использование bean-компонентов Spring в виде компонентов JMX MBean;
- наконец, глава 18 рассказывает, как развернуть приложение Spring в различных производственных окружениях, включая Kubernetes.

Разработчики, плохо знакомые с фреймворком Spring, должны начинать чтение с главы 1 и последовательно работать с каждой главой. Опытные разработчики могут предпочесть читать выборочно, переходя к наиболее интересным им темам. Однако каждая следующая глава строится на предыдущей, поэтому вы можете потерять нить рассуждений, начав чтение с середины книги.

О примерах программного кода

Книга содержит множество примеров программного кода и в виде листингов, и в виде отдельных фрагментов в тексте. Этот код всегда будет оформляться моноширинным шрифтом.

Во многих случаях исходный код был переформатирован: добавлены переносы строк и отступы, чтобы примеры уместились по ширине книжной страницы. В редких случаях этого оказалось недостаточно, поэтому там, где это необходимо, я добавил стрелки, обозначающие продолжение строки (➡). Кроме того, я удалил часть комментариев из кода, если он подробно описывается в тексте. Многие листинги сопровождаются дополнительным описанием, чтобы подчеркнуть наиболее важные идеи.

Выполнимые файлы примеров можно получить на странице онлайн-версии этой книги по адресу <https://livebook.manning.com/book/spring-in-action-sixth-edition>. Все примеры исходного кода доступны для загрузки на веб-сайте издательства Manning по адресу <https://www.manning.com/books/spring-in-action-sixth-edition>, а также в репозитории GitHub <http://github.com/habuma/spring-in-action-6-samples>.

Форум книги

Одновременно с покупкой «Spring в действии, шестое издание» вы получаете бесплатный доступ к liveBook – онлайн-платформе издательства Manning. Используя возможности liveBook, вы сможете оставлять свои комментарии к книге в целом или к определенным разделам или абзацам, делать заметки для себя, задавать технические вопросы и отвечать на них, а также получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, перейдите по ссылке <https://forums.manning.com/forums/spring-in-action-sixth-edition>. Дополнительные сведения о форумах Manning и правилах поведения на них можно получить по адресу <https://forums.manning.com/forums/about>.

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

Форум и архивы предыдущих обсуждений будут доступны на сайте издательства, пока книга находится в печати.

Другие онлайн-ресурсы

Нужна дополнительная помощь?

- На веб-сайте Spring есть несколько полезных руководств, описывающих, как начать работу (некоторые из них были написаны автором этой книги). Они доступны по адресу <https://spring.io/guides>.
- Сайт в Stack Overflow (теги *Spring* (<https://stackoverflow.com/questions/tagged/spring>) и *Spring Boot* (<https://stackoverflow.com/questions/tagged/springboot>)) – отличное место, где вы сможете задавать вопросы и помогать другим в освоении Spring. Помогать кому-то еще, отвечая на вопросы о Spring, – отличный способ изучения Spring!

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@

[gmail.com](mailto:dmkpress@gmail.com). Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Manning Publications очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Об авторе

Крейг Уоллс (Craig Walls) работает старшим инженером-разработчиком в VMware. Он настойчиво продвигает Spring Framework, часто выступает на встречах в местных группах пользователей и конференциях и пишет о Spring. Когда Крейг не пишет программный код, он обычно планирует свою следующую поездку в Диснейленд и проводит все свободное время со своей супругой, двумя дочерьми, тремя собаками и попугаем.

Об иллюстрации на обложке

На обложке «Spring в действии, шестое издание» изображен «Le Caraço» – житель района Карак на юго-западе Иордана. Столица Иордана – город Эль-Карак, который славится древним замком на холме с завораживающим видом на Мертвое море и окрестности. Рисунок взят из Французского туристического путеводителя «Encyclopedie des Voyages» (автор J. G. St. Saveur), выпущенного в 1796 году. Путешествия ради удовольствия были сравнительно новым явлением в то время, и туристические справочники, такие как этот, были популярны, они позволяли знакомиться с жителями других регионов Франции и других стран, не слезая с кресла.

Многообразие рисунков в путеводителе «Encyclopedie des Voyages» отчетливо демонстрирует уникальные и индивидуальные особенности городов и районов мира, существовавших 200 лет назад. Это было время, когда по одежде можно было отличить двух людей, проживающих в двух регионах, расположенных на расстоянии нескольких десятков миль. Туристический справочник позволяет почувствовать изолированность и отдаленность того периода от любого другого исторического периода, отличного от нашей гиперподвижной современности.

С тех пор мода изменилась, а региональные различия, такие существенные в те времена, исчезли. Сейчас зачастую сложно отличить жителей разных континентов. Возможно, пытаясь рассматривать это с оптимистической точки зрения, мы обменяли культурное и визуальное разнообразие на более разнообразную личную жизнь. Или более разнообразную и интересную интеллектуальную жизнь и техническую вооруженность.

Мы в издательстве Manning славим изобретательность, предпримчивость и радость компьютерного бизнеса обложками книг с рисунками из этого туристического справочника, изображающими богатство региональных различий двухвековой давности.

Часть I

Основы *Spring*

В части I этой книги вы начнете писать приложение с использованием Spring и попутно будете изучать основы данного фреймворка.

В главе 1 я дам краткий обзор основных принципов, лежащих в основе Spring и Spring Boot, и покажу, как создать проект приложения на Spring на примере создания Taco Cloud – вашего первого приложения на Spring. В главе 2 вы познакомитесь с Spring MVC и узнаете, как представлять модели данных в браузере и обрабатывать и проверять ввод в формах. Вы также получите несколько советов по выбору библиотеки шаблонов представлений. В главе 3 вы добавите в приложение Taco Cloud возможность сохранения данных, где также узнаете об использовании шаблона Spring JDBC и о том, как вставлять данные с помощью параметризованных запросов. Затем вы увидите, как объявлять репозитории JDBC (Java Database Connectivity) и JPA (Java Persistence API) с помощью Spring Data. Глава 4 продолжит рассказ о механизмах хранения данных в Spring и представит еще два модуля Spring Data для сохранения данных в Cassandra и MongoDB. Глава 5 посвящена вопросам обеспечения безопасности в приложениях Spring, включая автоматическую настройку Spring Security, определение пользовательского хранилища, настройку страницы входа и защиту от атак с подделкой межсайтовых запросов. В завершение части I, в главе 6, мы рассмотрим настройку конфигурационных свойств. Вы узнаете, как настраивать автоматически конфигурируемые bean-компоненты, как применять конфигурационные свойства к компонентам приложения и как работать с профилями Spring.

1

Введение в Spring

В этой главе рассматриваются следующие темы:

- основы Spring и Spring Boot;
- инициализация проекта Spring;
- обзор экосистемы Spring.

Греческий философ Гераклит не был известен как разработчик программного обеспечения, однако похоже, что он хорошо разбирался в этом вопросе. Вот его высказывание: «Единственная постоянная вещь – это перемены». Оно отражает фундаментальную истину разработки программного обеспечения.

Наши современные подходы к разработке приложений отличаются от подходов, использовавшихся 5, 10 и, конечно же, 20 лет тому назад, до того, как Spring Framework был представлен в книге Рода Джонсона (Rod Johnson) «Expert One-on-One J2EE Design and Development» (Wrox, 2002, <http://mng.bz/oVjy>).

В то время наиболее распространенными типами разрабатываемых приложений были браузерные веб-приложения, поддерживающие реляционные базы данных. Приложения этого типа по-прежнему актуальны – и Spring прекрасно подходит для их разработки, – однако в настоящее время нас больше интересует разработка приложений на основе микросервисов, предназначенных для размещения в облаке и хранящих данные в самых разных базах данных. А новый интерес к реактивному программированию направлен на обеспечение боль-

шей масштабируемости и улучшенной производительности за счет неблокирующих операций.

С развитием методик разработки программного обеспечения также менялся и Spring Framework. В нем появлялись все новые и новые средства, помогающие решать проблемы современной разработки, включая поддержку микросервисов и реактивное программирование. Кроме того, создатели Spring решили упростить модель разработки, внедрив Spring Boot.

Независимо от того, что разрабатывается – простое ли веб-приложение на основе базы данных или современное приложение на основе микросервисов, – Spring даст вам все и поможет достичь поставленных целей. Эта глава – ваш первый шаг в путешествии по современной разработке приложений с помощью Spring.

1.1 *Что такое Spring?*

Понимаю, что вам не терпится приступить к разработке с использованием Spring, и уверяю вас, что до окончания этой главы вы напишете простое приложение. Но сначала позвольте мне представить несколько основных идей, лежащих в основе Spring, которые помогут вам понять суть этого фреймворка.

Любое нетривиальное приложение состоит из множества компонентов, каждый из которых вносит свой вклад в общую функциональность, координируя свои действия с другими элементами приложения. Когда приложение запускается, эти компоненты должны как-то узнать о существовании друг друга.

По сути, Spring предлагает *контейнер*, часто называемый *контекстом приложения Spring*, который создает компоненты приложения и управляет ими. Эти компоненты, или *bean-компоненты*, объединяются внутри контекста Spring, образуя полноценное приложение, подобно тому, как кирпичи, известковый раствор, древесина, гвозди, водопроводные трубы и проводка соединяются вместе, образуя дом.

Акт объединения bean-компонентов основан на шаблоне, известном как *внедрение зависимостей* (Dependency Injection, DI). В технологии внедрения зависимостей компоненты не создают и не поддерживают жизненный цикл других компонентов, от которых они зависят, а полагаются в этом на отдельный объект (контейнер), который создает все нужные компоненты и внедрит их в другие компоненты, которые в них нуждаются. Обычно это делается с помощью аргументов конструктора или методов доступа к свойствам.

Например, предположим, что вам нужно обратиться к двум компонентам: службе инвентаризации (для получения информации о наличии запасов на складе) и службе продукта (за сведениями о продукте). Служба продукта зависит от службы инвентаризации, получая от нее полный набор информации о продуктах. Отношения между

этими компонентами и контекстом приложения Spring иллюстрирует рис. 1.1.

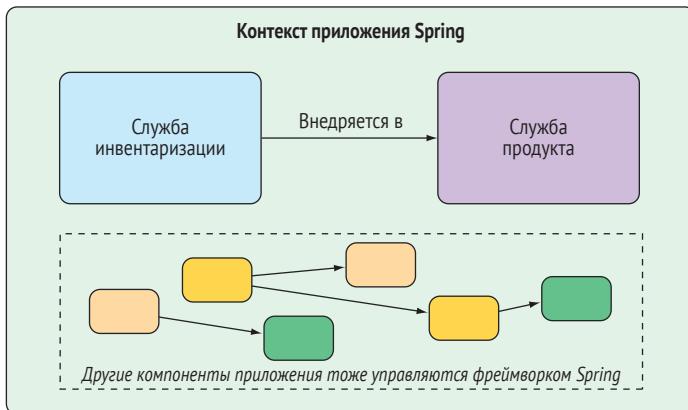


Рис. 1.1 Компоненты приложения управляются и внедряются друг в друга приложением контекста Spring

Помимо основного контейнера, Spring и сопутствующие библиотеки предлагают веб-фреймворк, различные механизмы хранения данных, фреймворк безопасности, интеграцию с другими системами, мониторинг времени выполнения, поддержку микросервисов, модель реактивного программирования и многое другое, необходимое для разработки современных приложений.

Исторически сложилось так, что способ управления контекстом приложения Spring для связывания bean-компонентов был основан на одном или нескольких XML-файлах, описывающих компоненты и их взаимосвязи с другими компонентами.

Например, следующий фрагмент XML объявляет два bean-компонента, `nventoryService` и `ProductService`, и внедряет `InventoryService` в `ProductService` через аргумент конструктора:

```
<bean id="inventoryService"
      class="com.example.InventoryService" />

<bean id="productService"
      class="com.example.ProductService" />
<constructor-arg ref="inventoryService" />
</bean>
```

Однако в последних версиях Spring принято производить такие настройки в коде на Java в специальном классе-конфигураторе. Следующий Java-класс описывает эквивалентную конфигурацию:

```
@Configuration
public class ServiceConfiguration {
    @Bean
```

```

public InventoryService inventoryService() {
    return new InventoryService();
}

@Bean
public ProductService productService() {
    return new ProductService(inventoryService());
}
}

```

Аннотация `@Configuration` подсказывает фреймворку Spring, что это класс конфигурации, который создает bean-компоненты для контекста Spring.

Методы класса конфигурации снабжены аннотацией `@Bean`, указывающей, что возвращаемые ими объекты должны быть добавлены в контекст приложения как bean-компоненты (где эти компоненты по умолчанию будут доступны по идентификаторам, совпадающим с именами определяющих их методов).

Определение конфигурации в Java-коде имеет определенные преимущества перед описанием в XML-файлах, в том числе более высокий уровень безопасности типов и простоту рефакторинга. Тем не менее явное описание конфигурации на Java или в XML необходимо, только если Spring не может автоматически настроить компоненты.

Автоматическая настройка уходит корнями в методы Spring, известные как *автоматическое связывание (autowiring)* и *сканирование компонентов*. Используя механизм сканирования, Spring может автоматически обнаруживать компоненты в пути поиска классов (`classpath`) приложения и создавать их как bean-компоненты в контексте приложения Spring. Механизм автоматического связывания позволяет фреймворку Spring внедрять компоненты в другие bean-компоненты, от которых те зависят.

Совсем недавно, с появлением Spring Boot, автоматическая настройка вышла далеко за рамки сканирования компонентов и автоматического связывания. Spring Boot – это расширение для Spring Framework, предлагающее несколько улучшений. Наиболее известным из них является *автоконфигурация* – Spring Boot может делать обоснованные предположения о том, какие компоненты следует настроить и связать вместе, опираясь на элементы в пути поиска классов, переменные окружения и другие факторы.

Я хотел бы показать вам пример кода, демонстрирующий автоконфигурацию, но не могу. Автоконфигурация похожа на ветер – вы можете видеть ее последствия, но у нее нет кода, который можно показать вам и сказать: «Смотрите! Вот пример автоконфигурации!» Механизм автоконфигурации просто работает, создает и связывает компоненты – и не требует для этого писать какой-либо код. Именно отсутствие необходимости писать код делает автоконфигурацию такой замечательной.

Автоконфигурация, предлагаемая Spring Boot, значительно сократила объем явного описания конфигурации (с помощью XML или на Java), необходимого для создания приложения. Фактически, закончив пример приложения в этой главе, вы получите действующее приложение Spring, содержащее только одну строку описания конфигурации!

Spring Boot настолько расширяет возможности разработки Spring, что без него трудно представить разработку Spring-приложений. По этой причине мы будем рассматривать Spring и Spring Boot как одно и то же. Мы будем максимально использовать Spring Boot, а явное описание конфигурации будем добавлять только там, где это действительно необходимо. А поскольку описание конфигурации в XML – это старый способ работы со Spring, мы будем описывать конфигурацию приложений Spring почти исключительно в коде на Java.

Но довольно болтовни. В названии этой книги есть слова «в действии», так что давайте будем действовать и начнем писать наше первое приложение на Spring.

1.2 Инициализация приложения Spring

На протяжении всей книги мы с вами будем создавать онлайн-приложение Taco Cloud для заказа самой замечательной еды, созданной человеком, – тако. Конечно, для достижения этой цели мы будем использовать Spring, Spring Boot и множество связанных с ними библиотек и фреймворков.

Инициализировать проекты приложений Spring можно несколькими способами. Я мог бы показать вам все этапы создания структуры каталогов проекта и определения спецификации сборки вручную, но, по большому счету, это пустая трата времени, которое лучше потратить на написание кода. Поэтому мы доверимся Spring Initializr.

Spring Initializr – это веб-приложение, помогающее создать скелетную структуру проекта Spring, которую вы затем сможете наполнить любой функциональностью, какой захотите. Вот несколько способов использования Spring Initializr:

- открыть в браузере страницу <http://start.spring.io>;
- обратиться к приложению с помощью утилиты curl;
- воспользоваться интерфейсом командной строки Spring Boot;
- создать новый проект с помощью Spring Tool Suite;
- создать новый проект с помощью IntelliJ IDEA;
- создать новый проект с помощью Apache NetBeans.

Чтобы не тратить время на обсуждение каждого из этих вариантов в этой главе, я вынес все детали в приложение. А в этой главе и на протяжении всей книги я буду показывать, как создать новый проект, используя мой любимый вариант: с помощью поддержки Spring Initializr в Spring Tool Suite.

Как следует из названия, Spring Tool Suite – это среда разработки (Integrated Development Environment, IDE) на Spring, которая поставляется в форме расширений для Eclipse, Visual Studio Code или Theia IDE. Готовые к использованию двоичные файлы Spring Tool Suite можно получить по адресу <https://spring.io/tools>. Spring Tool Suite предлагает удобный инструмент Spring Boot Dashboard, позволяющий легко запускать, перезапускать и останавливать приложения Spring Boot из среды разработки.

Даже если вы не хотите использовать Spring Tool Suite, ничего страшного, мы все равно можем остаться друзьями. Перейдите к приложению в конце книги и выберите вариант, который вам больше по душе. Но знайте, что в этой книге я могу время от времени упоминать инструменты, имеющиеся только в Spring Tool Suite, такие как Spring Boot Dashboard. Если вы не используете Spring Tool Suite, то вам придется адаптировать мои инструкции к вашей среде разработки.

1.2.1 Инициализация проекта Spring с помощью Spring Tool Suite

Чтобы начать работу над новым проектом Spring в Spring Tool Suite, откройте меню **File** (Файл) и выберите пункт **New** (Создать), а затем **Spring Starter Project** (Новый проект Spring). На рис. 1.2 для ясности показана структура меню.

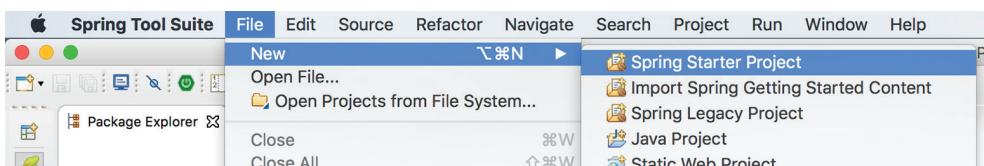


Рис. 1.2 Создание нового проекта с помощью Initializr в Spring Tool Suite

После выбора пункта меню **Spring Starter Project** (Новый проект Spring) появится окно мастера создания нового проекта (рис. 1.3). На первой странице мастер попросит ввести некоторую общую информацию о проекте: имя проекта, описание и т. д. Если вы знакомы с содержимым файла pom.xml для Maven, то в большинстве полей без труда опознаете элементы спецификации сборки Maven. Для приложения Taco Cloud заполните поля, как показано на рис. 1.3, и щелкните на кнопке **Next** (Далее).

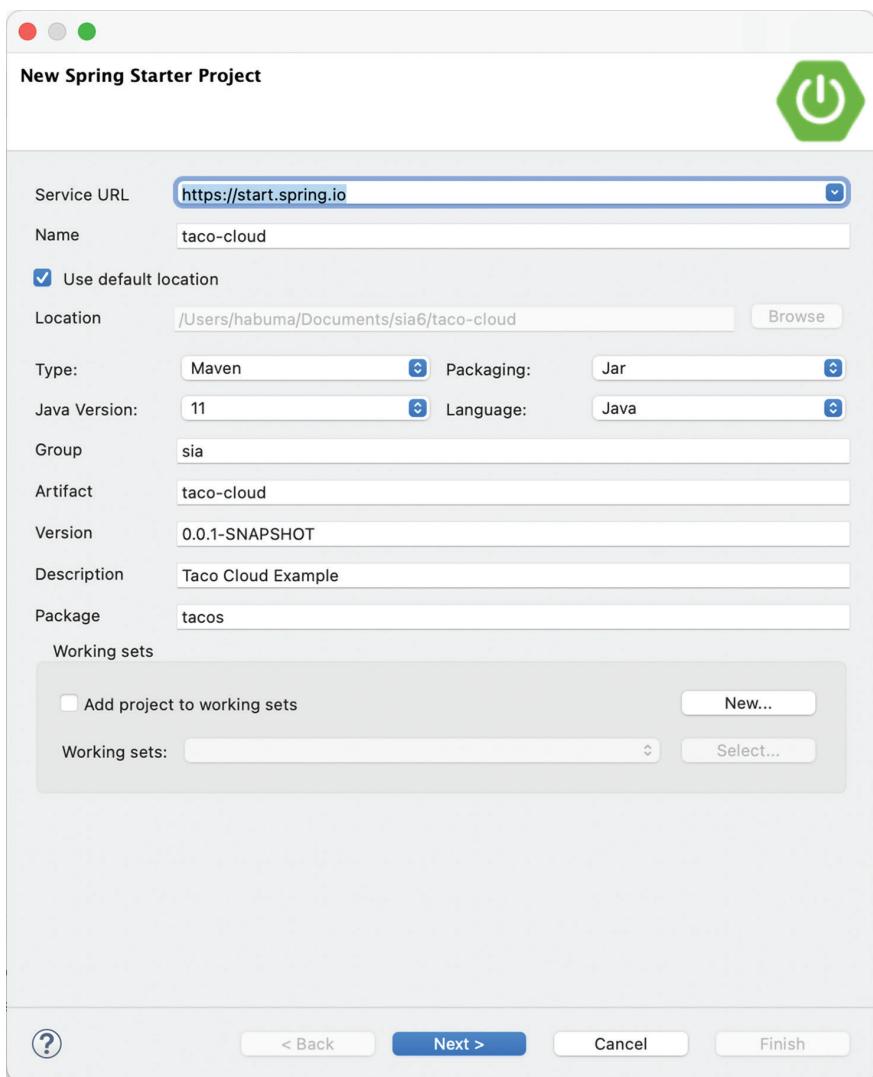


Рис. 1.3 Заполните поля информацией о проекте приложения Taco Cloud

На следующей странице мастера нужно выбрать зависимости для проекта (рис. 1.4). Обратите внимание, что в верхней части диалога можете указать версию Spring Boot, на которой должен основываться проект. По умолчанию используется самая последняя доступная версия. Как правило, рекомендуется оставлять значения по умолчанию, если только вам не нужна какая-то конкретная версия.

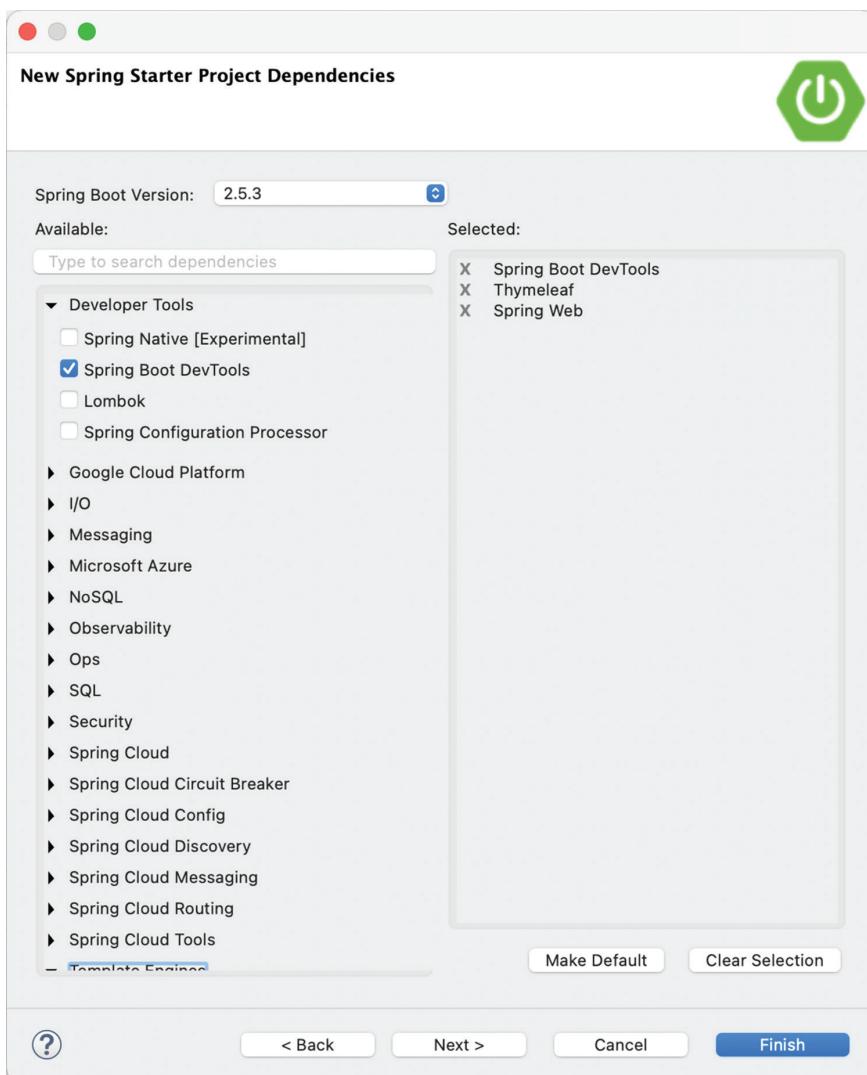


Рис. 1.4 Выбор зависимостей для проекта

При настройке зависимостей можно распахивать различные разделы и выбирать нужные зависимости вручную или искать их с помощью поля **Available** (Доступно) вверху. Для приложения Taco Cloud выберите зависимости, как показано на рис. 1.4.

Теперь можно щелкнуть на кнопке **Finish** (Готово), чтобы сгенерировать проект и добавить его в свое рабочее пространство. Но если вам хочется приключений, щелкните на кнопке **Next** (Далее) еще раз, чтобы увидеть последнюю страницу мастера создания нового проекта, как показано на рис. 1.5.

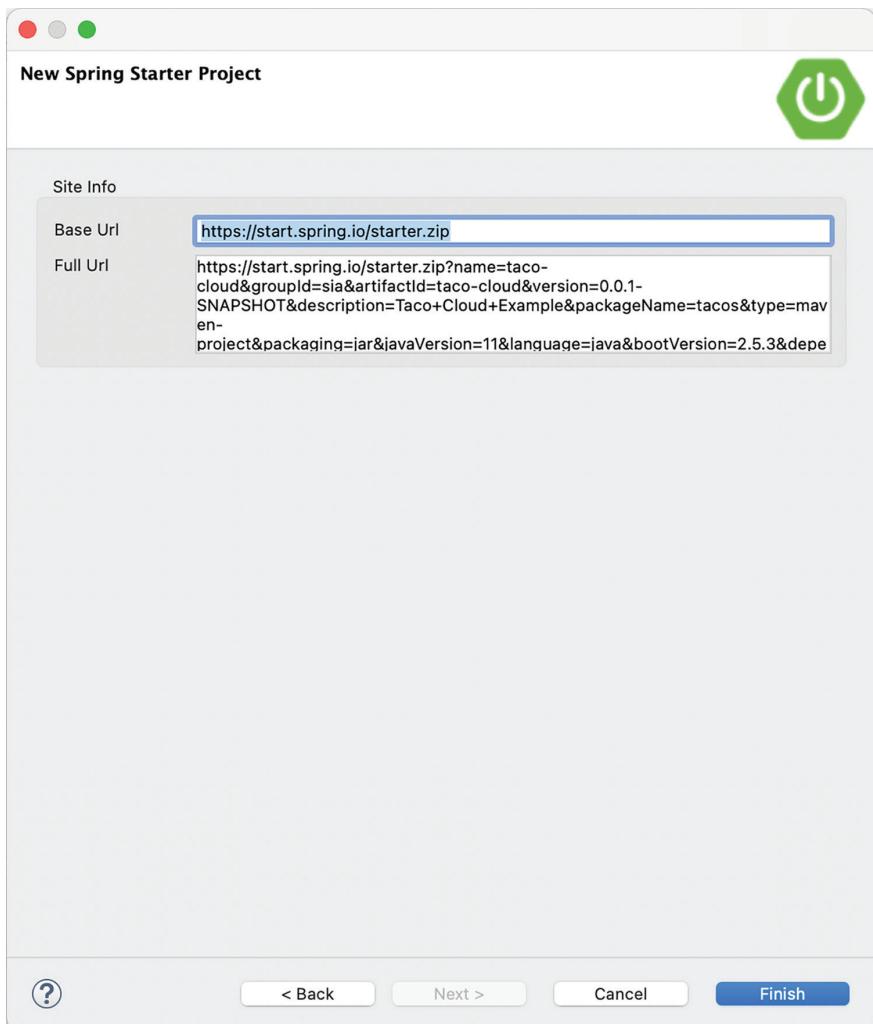


Рис. 1.5 Ввод необязательного альтернативного адреса Initializr

По умолчанию мастер создания нового проекта вызывает веб-приложение Spring Initializr, обращаясь по адресу <http://start.spring.io>. Обычно нет необходимости переопределять это значение по умолчанию, поэтому можно смело щелкнуть на кнопке **Finish** (Готово) на второй странице мастера. Но если по какой-то причине вы вынуждены использовать свой клон Initializr (локальную копию на своем компьютере или настроенный клон, работающий в локальной сети вашей компании), то вы можете изменить поле **Base Url** (Базовый URL), подставив адрес вашего экземпляра Initializr.

После щелчка на кнопке **Finish** (Готово) проект будет загружен из Initializr и размещен в вашей рабочей области. Подождите несколько

секунд, пока закончится загрузка и сборка, и после этого вы будете готовы приступить к разработке приложения. Но сначала давайте посмотрим, что нам дал Initializr.

1.2.2 Структура проекта Spring

После загрузки проекта в среду разработки распахните его, чтобы увидеть содержимое. На рис. 1.6 показано содержимое проекта Taco Cloud в Spring Tool Suite.

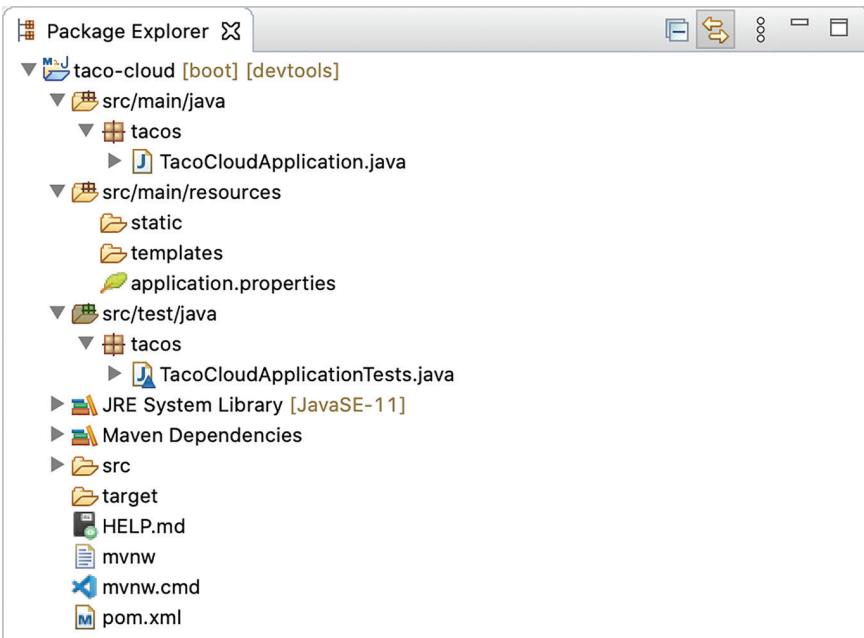


Рис. 1.6 Начальная структура проекта в Spring Tool Suite

Как видите, проект имеет типичную структуру проекта Maven или Gradle, в которой исходный код помещается в каталог *src/main/java*, код тестов – в *src/test/java*, а ресурсы, не являющиеся Java-ресурсами, – в *src/main/resources*. Обратите внимание на следующие элементы внутри этой структуры:

- *mvnw* и *mvnw.cmd* – это сценарии-обертки Maven, эти сценарии можно использовать для создания нового проекта, даже если на вашем компьютере не установлен Maven;
- *pom.xml* – это параметры сборки Maven, вскоре мы рассмотрим ее поближе;
- *TacoCloudApplication.java* – это основной класс Spring Boot, который запускает проект, вскоре мы уделим ему наше внимание;
- *application.properties* – это файл, изначально пустой, используется для определения конфигурационных свойств. Мы не будем

особенно углубляться в него в данной главе, но подробно обсудим в главе 6;

- *static* – в эту папку можно поместить любой статический контент (изображения, таблицы стилей, JavaScript и т. д.), который должен отображаться в браузере. Изначально в этой папке ничего нет;
- *templates* – тут мы разместим файлы шаблонов, используемые для отображения контента в браузере. Изначально в этой папке ничего нет, но скоро мы добавим в нее шаблон Thymeleaf;
- *TacoCloudApplicationTests.java* – это простой тестовый класс, проверяющий успех загрузки контекста приложения Spring. Мы будем добавлять в него дополнительные тесты по мере разработки приложения.

Развивая приложение Taco Cloud, мы будем заполнять эту базовую структуру проекта кодом на Java, изображениями, таблицами стилей, тестами и другими необходимыми ресурсами. А пока поближе рассмотрим некоторые элементы, которые создало веб-приложение Spring Initializr.

ПАРАМЕТРЫ СБОРКИ

Заполняя форму Initializr, мы указали, что сборка проекта будет производиться с помощью Maven. Поэтому приложение Spring Initializr создало файл *pom.xml*, уже заполненный необходимыми параметрами. Его содержимое показано в листинге 1.1.

Листинг 1.1 Параметры сборки для 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  
        https://maven.apache.org/xsd/maven-4.0.0.xsd">  
    <modelVersion>4.0.0</modelVersion>  
    <parent>  
        <groupId>org.springframework.boot</groupId>  
        <artifactId>spring-boot-starter-parent</artifactId>  
        <version>2.5.3</version> ← Версия Spring Boot  
        <relativePath />  
    </parent>  
    <groupId>sia</groupId>  
    <artifactId>taco-cloud</artifactId>  
    <version>0.0.1-SNAPSHOT</version>  
    <name>taco-cloud</name>  
    <description>Taco Cloud Example</description>  
  
    <properties>  
        <java.version>11</java.version>  
    </properties>  
  
    <dependencies> ← Зависимости инструмента Starter  
        <dependency>
```

```

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

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

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <scope>runtime</scope>
    <optional>true</optional>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <groupId>org.junit.vintage</groupId>
            <artifactId>junit-vintage-engine</artifactId>
        </exclusion>
    </exclusions>
</dependency>

</dependencies>

<build>
    <plugins>
        <plugin> ←———— Плагин Spring Boot
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

<repositories>
    <repository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
    </repository>
</repositories>

<pluginRepositories>
    <pluginRepository>
        <id>spring-milestones</id>
        <name>Spring Milestones</name>
        <url>https://repo.spring.io/milestone</url>
    </pluginRepository>
</pluginRepositories>
</project>

```

Первое, на что следует обратить внимание, – это элемент `<parent>` и его дочерний элемент `<version>`. Он сообщает, что ваш проект имеет родительский файл POM¹ `spring-boot-starter-parent`. Помимо прочего, родительский POM определяет зависимости от нескольких библиотек, часто используемых в проектах Spring. Для библиотек, определяемых в родительском POM, не нужно указывать версию, потому что она наследуется от родителя. Версия 2.5.6 соответствует Spring Boot 2.5.6, соответственно, управление зависимостями осуществляется, как определено в этой версии Spring Boot. Среди прочего Spring Boot 2.5.6 определяет базовую версию ядра Spring Framework – 5.3.12.

Поскольку речь зашла о зависимостях, обратите внимание на четыре зависимости, объявленные в элементе `<dependencies>`. Первые три должны показаться вам знакомыми. Они напрямую соответствуют зависимостям Spring Web, Thymeleaf и Spring Boot DevTools, которые мы выбрали в мастере создания нового проекта Spring Tool Suite. Четвертая зависимость подключает средства тестирования. Чтобы ее включить, не нужно ставить галочку, потому что Spring Initializr предполагает (надеюсь, правильно), что вы будете писать тесты.

Также обратите внимание, что все зависимости, кроме DevTools, имеют слово *starter* в идентификаторе артефакта. Зависимости от Spring Boot Starter отличаются тем, что они обычно не имеют своего библиотечного кода, а транзитивно подключают другие библиотеки. Эти зависимости предлагают следующие основные преимущества:

- благодаря им файл сборки получится меньше и им будет легче управлять, поскольку отпадает необходимость объявлять зависимости для каждой библиотеки, которая вам может понадобиться;
- зависимости можно рассматривать с точки зрения предоставляемых ими возможностей, а не с точки зрения имен их библиотек. Разрабатывая веб-приложение, вам достаточно добавить зависимость `spring-boot-starter-web` вместо длинного списка библиотек, которые позволят вам написать веб-приложение;
- освобождают от беспокойства о версиях библиотек. Вы можете быть уверены, что версии библиотек, добавленных транзитивно, будут совместимы с данной версией Spring Boot. Вам остается только определиться с версией Spring Boot.

В конце файла с параметрами сборки определяется плагин Spring Boot. Он выполняет несколько важных функций, в том числе:

- определяет цель Maven, которая позволяет запускать приложение с помощью Maven;

¹ POM (Project Object Model – объектная модель проекта) – базовый модуль Maven, специальный XML-файл, который всегда хранится в базовом каталоге проекта и имеет имя `pom.xml`. Файл POM содержит информацию о проекте и различных параметрах конфигурации, которые используются Maven для сборки.

- гарантирует включение в выполняемый файл JAR всех библиотек-зависимостей и их доступность в пути поиска классов class-path во время выполнения;
- создает файл манифеста в архиве JAR, который определяет класс начальной загрузки (в вашем случае TacoCloudApplication) в качестве главного класса для выполняемого файла JAR.

Давайте рассмотрим класс начальной загрузки поближе.

ЗАГРУЗКА ПРИЛОЖЕНИЯ

Поскольку приложение будет запускаться из выполняемого JAR-файла, важно выбрать главный класс, который будет выполняться при запуске этого JAR-файла. Нам также потребуется хотя бы минимальная конфигурация Spring для начальной загрузки приложения. Все это находится в классе TacoCloudApplication, показанном в листинге 1.2.

Листинг 1.2 Класс начальной загрузки для проекта Taco Cloud

```
package tacos;

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

@SpringBootApplication ← Приложение Spring Boot
public class TacoCloudApplication {

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

Несмотря на небольшой размер, класс TacoCloudApplication играет очень важную роль. Одна из самых важных строк – одновременно одна из самых коротких. Аннотация @SpringBootApplication ясно указывает, что это приложение Spring Boot. Но этим функции аннотации @SpringBootApplication не ограничиваются.

@SpringBootApplication – это составная аннотация, объединяющая три другие аннотации:

- *@SpringBootConfiguration* – определяет этот класс как класс конфигурации. В данный момент в этом классе не определяется никаких конфигурационных параметров, но если понадобится, в него можно добавить настройки Spring Framework. Эта аннотация, по сути, является специализированной формой аннотации *@Configuration*;
- *@EnableAutoConfiguration* – включает автоконфигурацию Spring Boot. Подробнее об автоконфигурации мы поговорим позже, а пока просто имейте в виду, что эта аннотация сообщает Spring Boot о необходимости автоматически настраивать любые компоненты, которые могут вам понадобиться;

- `@ComponentScan` – включает сканирование компонентов. Механизм сканирования позволяет объявлять другие классы с аннотациями, такими как `@Component`, `@Controller` и `@Service`, чтобы фреймворк Spring автоматически обнаруживал и регистрировал их как компоненты в контексте приложения Spring.

Другой важной частью `TacoCloudApplication` является метод `main()`. Этот метод будет вызываться в момент запуска файла JAR. В большинстве приложений данный метод содержит шаблонный код; каждое написанное вами приложение Spring Boot будет иметь метод `main()`, аналогичный или идентичный этому (несмотря на различия в именах классов).

Метод `main()` вызывает статический метод `run()` класса `SpringApplication`, который выполняет фактическую загрузку приложения, создавая контекст приложения Spring. Метод `run()` принимает два параметра: класс конфигурации и аргументы командной строки. В общем случае совсем необязательно, чтобы имя класса конфигурации, передаваемого в вызов `run()`, совпадало с именем класса начальной загрузки, но это наиболее удобный и распространенный выбор.

В большинстве случаев вам не придется ничего менять в классе начальной загрузки. Для простых приложений может быть удобно настроить один или два других компонента в классе начальной загрузки, но для большинства приложений лучше создать отдельный класс конфигурации и настраивать в нем все, что не настраивается автоматически. В этой книге мы определим несколько классов конфигурации, так что не теряйте нить рассуждений.

ТЕСТИРОВАНИЕ ПРИЛОЖЕНИЯ

Тестирование – важная часть разработки программного обеспечения. Вы всегда можете протестировать свой проект вручную, создав его, а затем запустив тестирование из командной строки:

```
$ ./mvnw package  
...  
$ java -jar target/taco-cloud-0.0.1-SNAPSHOT.jar
```

Или, благодаря использованию Spring Boot, плагин Spring Boot Maven упрощает этот шаг еще больше:

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

Но тестирование вручную подразумевает участие человека и, как следствие, вероятность потенциальных человеческих ошибок и не-последовательность тестирования. Автоматизированные тесты более последовательны и воспроизводимы.

Учитывая это, Spring Initializr предоставляет заготовку тестового класса, как показано в листинге 1.3.

Листинг 1.3 Заготовка класса для тестирования приложения

```
package tacos;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;
@SpringBootTest
public class TacoCloudApplicationTests {
    @Test
    public void contextLoads() {
    }
}
```

В TacoCloudApplicationTests практически ничего нет: единственный тестовый метод в классе пуст. И все же этот тестовый класс выполняет важную проверку – успешность загрузки контекста приложения Spring. Если вы внесете какие-либо изменения, препятствующие созданию контекста приложения Spring, то этот тест завершится ошибкой, и вы сможете отреагировать на нее, устранив проблему.

Аннотация `@SpringBootTest` настраивает JUnit на запуск теста с поддержкой возможностей Spring Boot. Подобно `@SpringBootApplication`, `@SpringBootTest` – это составная аннотация, которая сама снабжена аннотацией `@ExtendWith(SpringExtension.class)`, добавляющей поддержку возможностей тестирования Spring в JUnit 5. Однако вы пока можете думать о ней как об эквиваленте вызова `SpringApplication.run()` в методе `main()`. В этой книге вы несколько раз встретитесь с аннотацией `@SpringBootTest` и мы еще будем обсуждать некоторые ее возможности.

Наконец, в классе есть тестовый метод. Аннотация `@SpringBootTest` выполняет загрузку контекста приложения Spring для теста, но сам тестовый класс ничего не будет делать, если в нем не будет тестовых методов. Даже в таком виде, без всяких проверок и любого другого кода, этот пустой тестовый метод выполнит свою работу за счет аннотаций и загрузит контекст приложения Spring. Если при этом возникнут какие-либо проблемы, тест сообщит об ошибке.

Этот и любые другие тестовые классы можно запустить из командной строки, используя следующее заклинание Maven:

```
$ ./mvnw test
```

На этом мы завершаем обзор кода, генерированного веб-приложением Spring Initializr. Вы познакомились с основой, которую можно использовать для разработки приложения Spring, но до сих пор не написали ни строчки кода. Теперь пришло время запустить вашу среду разработки, стряхнуть пыль с клавиатуры и добавить свой код в приложение Taco Cloud.

1.3 Разработка приложения Spring

Поскольку вы только в начале пути, начнем с относительно небольшого изменения в приложении Taco Cloud, но оно продемонстрирует многие достоинства Spring. Вполне уместно, если первым вашим шагом – первой функцией в приложении Taco Cloud – станет домашняя страница. Для этого мы добавим следующие два артефакта:

- класс контроллера, обрабатывающий запросы к домашней странице;
- шаблон представления, определяющий внешний вид домашней страницы.

А поскольку в разработке ПО важную роль играет тестирование, мы напишем также простой тестовый класс для проверки домашней страницы. Но обо всем по порядку... Начнем с класса контроллера.

1.3.1 Обработка веб-запросов

В состав Spring входит мощная веб-платформа, известная как Spring MVC. В основе Spring MVC лежит идея *контроллера* – класса, обрабатывающего запросы и возвращающего некоторую информацию. В веб-приложениях контроллер отвечает, заполняя, при необходимости, модель данных и передавая запрос представлению для создания разметки HTML, которая возвращается браузеру.

Подробнее о Spring MVC мы поговорим в главе 2, а пока просто напишем простой класс контроллера, который обрабатывает запросы с корневым путем (например, /) и передает их представлению домашней страницы без заполнения модели данных. В листинге 1.4 показан простой класс контроллера.

Листинг 1.4 Контроллер домашней страницы

```
package tacos;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller           | Контроллер
public class HomeController {

    @GetMapping("/")   | Обрабатывает запросы
    public String home() {           | с корневым путем /
        return "home";             | Возвращает имя
    }                           | представления
}
```

Как видите, этот класс снабжен аннотацией `@Controller`. Сама по себе эта аннотация почти ничего не делает. Ее основная цель – идентифицировать класс как компонент, доступный механизму сканиро-

вания компонентов. Поскольку `HomeController` снабжен аннотацией `@Controller`, механизм сканирования в Spring автоматически обнаружит его и создаст экземпляр `HomeController` как bean-компонент в контексте приложения Spring.

Той же цели служат еще несколько аннотаций (включая `@Component`, `@Service` и `@Repository`). Мы могли бы аннотировать класс `HomeController` любой из этих других аннотаций, и он все равно работал бы так же. Однако аннотация `@Controller` лучше описывает роль этого компонента в приложении.

Метод `home()` так же прост, как всякие другие методы контроллера. Он снабжен аннотацией `@GetMapping`, указывающей, что этот метод обрабатывает HTTP-запросы GET с корневым путем `/`. При этом он ничего не делает, кроме как возвращает строковое значение `home`.

Это значение интерпретируется как логическое имя представления. Особенности реализации этого представления зависят от нескольких факторов, но поскольку Thymeleaf находится в пути поиска классов, мы можем определить этот шаблон с помощью Thymeleaf.

Почему Thymeleaf?

Возможно, вам интересно, почему из всех механизмов шаблонов я выбрал именно Thymeleaf. Почему не JSP? Почему не FreeMarker? Почему не любой другой из множества вариантов?

В моем выборе нет потаенного смысла, я должен был что-то выбрать, а мне нравится Thymeleaf, и обычно я предпочитаю его другим вариантам. Многим очевидным выбором может показаться JSP, однако при использовании JSP с фреймворком Spring Boot необходимо преодолеть некоторые проблемы, а я не хотел спускаться в кроличью нору в главе 1. Но не переживайте, мы рассмотрим другие варианты механизмов шаблонов, включая JSP, в главе 2.

Имя шаблона состоит из имени логического представления, к которому добавляются префикс пути `/templates/` и расширение `.html`. В результате полный путь к шаблону будет иметь вид: `/templates/home.html`. Это означает, что шаблон должен находиться в папке проекта `/src/main/resources/templates/home.html`. Давайте создадим этот шаблон сейчас.

1.3.2 *Определение представления*

Наша домашняя страница будет простой и содержать лишь приветствие посетителя сайта. В листинге 1.5 показан простой шаблон Thymeleaf, определяющий домашнюю страницу Taco Cloud.

Листинг 1.5 Шаблон домашней страницы Taco Cloud

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Taco Cloud</title>
    </head>
    <body>
        <h1>Welcome to...</h1>
        
    </body>
</html>
```

Здесь почти нечего обсуждать. Единственная примечательная строка – строка с тегом ``, отображающая логотип Taco Cloud. В ней присутствует Thymeleaf-атрибут `th:src` и выражение `@{...}`, описывающее ссылку на изображение с контекстно-зависимым путем. В остальном это самая обычная страница Hello World.

Давайте поговорим немного об изображении логотипа. Вы можете определить свое изображение логотипа. Но вы должны поместить его в нужное место в проекте.

На изображение ссылается контекстно-зависимый путь `/images/TacoCloud.png`. Как вы помните из нашего обзора структуры проекта, статический контент, такой как изображения, хранится в папке `/src/main/resources/static`. То есть изображение логотипа Taco Cloud также должно находиться в проекте в пути `/src/main/resources/static/images/TacoCloud.png`.

Теперь, когда у нас есть контроллер для обработки запросов к домашней странице и шаблон представления для ее отображения, мы почти готовы запустить приложение и увидеть его в действии. Но сначала посмотрим, как написать тест для контроллера.

1.3.3 Тестирование контроллера

Тестирование веб-приложений может быть сложной задачей, потому что иногда требует проверки содержимого HTML-страницы. К счастью, Spring имеет мощные средства тестирования, упрощающие тестирование веб-приложений.

Для нашей домашней страницы мы напишем тест, сравнимый по сложности с самой домашней страницей. Наш тест (листинг 1.6) выполнит HTTP-запрос GET с корневым путем `/` и затем убедится, что контроллер выбрал имя представления `home`, а сама страница содержит фразу «Welcome to...».

Листинг 1.6 Тест для контроллера домашней страницы

```

package tacos;

import static org.hamcrest.Matchers.containsString;
import static
    org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;
import static
    org.springframework.test.web.servlet.result.MockMvcResultMatchers.view;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.web.servlet.WebMvcTest;
import org.springframework.test.web.servlet.MockMvc;

@WebMvcTest(HomeController.class) ← Тест для HomeController
public class HomeControllerTest {

    @Autowired
    private MockMvc mockMvc; ← Внедрить MockMvc

    @Test
    public void testHomePage() throws Exception {
        mockMvc.perform(get("/")) ← Выполнить запрос GET /
            .andExpect(status().isOk()) ← Ожидается код ответа HTTP 200
            .andExpect(view().name("home")) ← Ожидается имя представления home
            .andExpect(content().string(
                containsString("Welcome to..."))); ← Ожидается наличие строки
        } ← «Welcome to...»
    }
}

```

Первое, что бросается в глаза, – этот тест немного отличается от класса TacoCloudApplicationTests применяемыми к нему аннотациями. Вместо @SpringBootTest класс HomeControllerTest снабжен аннотацией @WebMvcTest. Это специальная тестовая аннотация из Spring Boot, которая организует запуск теста в контексте приложения Spring MVC. В данном случае она обеспечивает регистрацию класса HomeController в Spring MVC, чтобы дать возможность отправлять ему запросы.

@WebMvcTest также настраивает поддержку тестирования Spring MVC в Spring. Для тестирования можно было бы запустить сервер, но для наших целей вполне достаточно имитировать механику Spring MVC. В тестовый класс внедряется объект MockMvc, чтобы тест мог управлять фиктивным объектом.

Метод testHomePage() определяет тест для проверки домашней страницы. Он начинается с вызова объекта MockMvc для выполнения HTTP-запроса GET с / (корневым путем). В числе результатов выполнения этого запроса ожидается следующее:

- ответ должен иметь статус HTTP 200 (OK);
- представление должно иметь логическое имя `home`;
- получившаяся страница должна содержать текст «Welcome to...».

Тест можно запустить в среде разработки или с помощью Maven:

```
$ mvnw test
```

Если после выполнения запроса объектом `MockMvc` какое-либо из этих ожиданий не будет выполнено, тест завершится неудачей. Но наш контроллер и шаблон представления написаны так, чтобы удовлетворить эти ожидания, поэтому тест должен выполниться успешно или, по крайней мере, с некоторым оттенком зеленого, указывающим на успешное прохождение.

Контроллер написан, шаблон представления создан, и тест пройден. Похоже, мы успешно справились с реализацией домашней страницы. Но, кроме успешного прохождения теста, есть еще один приятный момент – наглядный результат в браузере. В конце концов, именно его увидят клиенты Taco Cloud. А теперь соберем приложение и запустим его.

1.3.4 Сборка и запуск приложения

У нас есть не только несколько способов инициализации приложения Spring, но и несколько способов запустить его. Если хотите, вы можете перейти к приложению в конце книги и прочитать о некоторых наиболее распространенных способах запуска приложений Spring Boot.

Поскольку мы решили использовать Spring Tool Suite для инициализации проекта и работы над ним, то воспользуемся удобным инструментом под названием Spring Boot Dashboard, который поможет запустить приложение в среде разработки. Spring Boot Dashboard отображается в виде вкладки, обычно в левом нижнем углу окна IDE. На рис. 1.7 показан скриншот Spring Boot Dashboard.

Я не буду тратить много времени на изучение всего, что делает Spring Boot Dashboard, хотя на рис. 1.7 показаны некоторые из наиболее полезных деталей. Сейчас важно знать, как использовать этот инструмент для запуска приложения Taco Cloud. Выберите приложение `taco-cloud` в списке проектов (это единственное приложение, показанное на рис. 1.7), а затем щелкните на кнопке запуска (крайняя левая кнопка с зеленым треугольником и красным квадратом). Приложение должно запуститься сразу после щелчка.

Когда приложение запустится, вы увидите, как в консоли пролетит некоторый логотип Spring, оформленный символами ASCII, а за ним последуют отладочные строки, описывающие шаги, выполняемые при запуске приложения. В их числе вы увидите строку, сообщающую, что сервер Tomcat запущен и прослушивает порт 8080 (`http`). Это означает, что теперь можно запустить веб-браузер и открыть домашнюю страницу приложения, чтобы увидеть плоды своего труда.

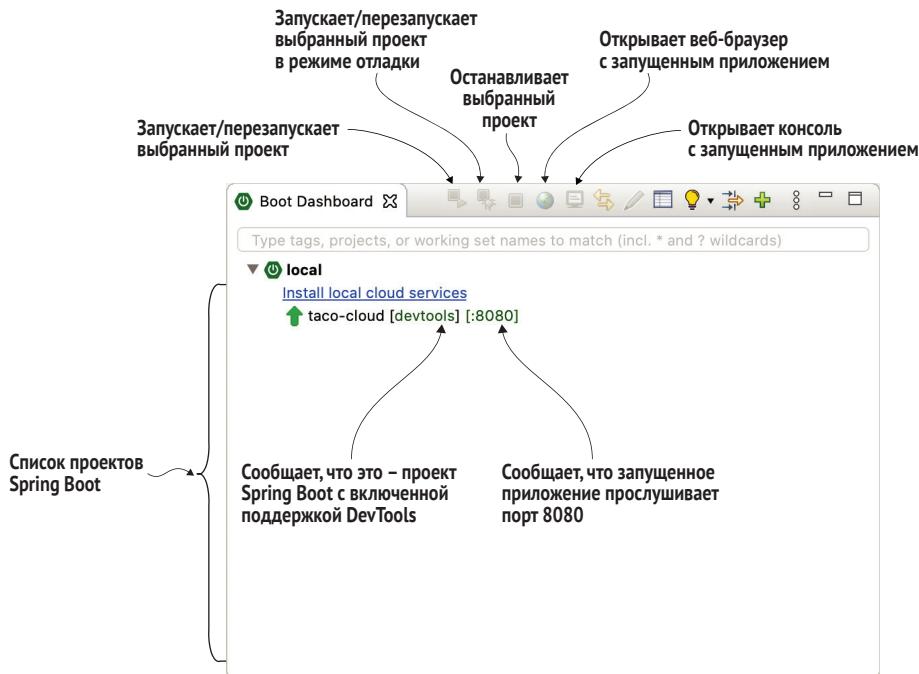


Рис. 1.7 Spring Boot Dashboard

Но постойте! Tomcat запустился? Когда мы успели развернуть приложение на веб-сервере Tomcat?

Приложения Spring Boot, как правило, несут в себе все, что им нужно, и не требуют развертывания на каком-либо сервере приложений. Мы не развертывали свое приложение на Tomcat – Tomcat является частью нашего приложения! (Я подробно опишу, как Tomcat стал частью приложения, в разделе 1.3.6.)

Теперь, когда приложение запущено, введите в веб-браузере адрес <http://localhost:8080> (или щелкните на кнопке с изображением глобуса на панели инструментов Spring Boot), и вы должны увидеть страницу, как показано на рис. 1.8. У вас страница может отличаться, если вы использовали свое изображение логотипа, но в остальном она не должна отличаться от рис. 1.8.

Страница выглядит незамысловато. Но и эта книга посвящена не графическому дизайну. На данный момент такой скромной домашней страницы более чем достаточно. И она дает нам хорошую отправную точку для дальнейшего знакомства со Spring.

Единственное, что я упустил из виду, – это набор инструментов разработчика DevTools. Мы выбрали его как зависимость при инициализации вашего проекта, и он отображается как зависимость в сгенерированном файле *pom.xml*. А панель Spring Boot Dashboard даже показывает, что в проекте включена поддержка DevTools. Но что это за инструменты DevTools и что они нам дают? Давайте кратко рассмотрим некоторые наиболее полезные функции DevTools.

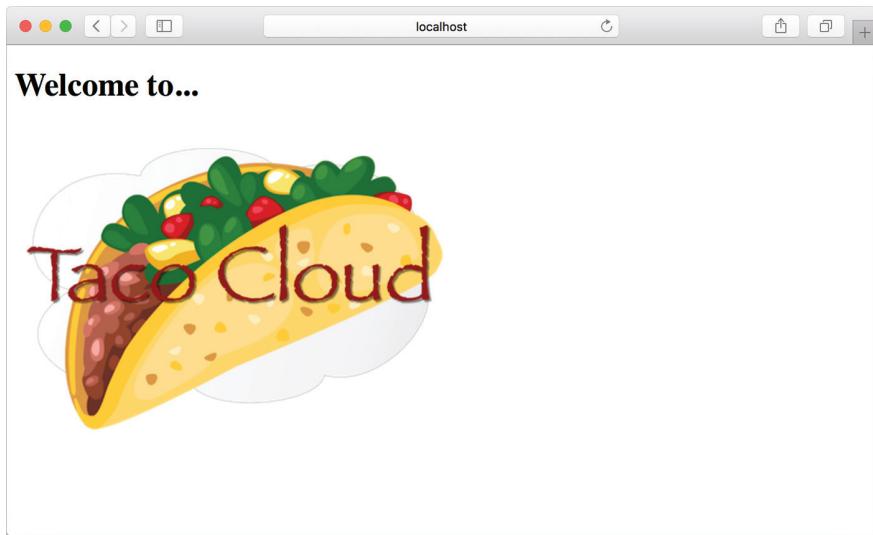


Рис. 1.8 Домашняя страница Taco Cloud

1.3.5 *Spring Boot DevTools*

Как следует из названия, DevTools – это набор дополнительных инструментов разработчика приложений на Spring, позволяющих:

- автоматически перезапускать приложение при изменении кода;
- автоматически обновлять окно браузера при изменении ресурсов, передаваемых браузеру (таких как шаблоны, сценарии на JavaScript, таблицы стилей и т. д.);
- автоматически отключать кеширование шаблонов;
- открывать встроенную консоль H2, если используется база данных H2.

Важно понимать, что DevTools не является плагином IDE и не требует использования конкретной IDE. Эти инструменты одинаково хорошо работают в Spring Tool Suite, IntelliJ IDEA и NetBeans. Кроме того, будучи предназначенными для разработки, они достаточно интеллектуальны, чтобы отключать себя при развертывании в промышленном окружении. Мы обсудим, как это делается, когда приступим к развертыванию приложения в главе 18. А пока сосредоточимся на наиболее полезных функциях Spring Boot DevTools и начнем с автоматического перезапуска приложения.

Автоматический перезапуск приложения

Наличие DevTools в составе проекта позволяет вносить изменения в код на Java и в файлы свойств проекта и тут же видеть, как эти изменения вступают в силу. DevTools следит за изменениями в файлах и, когда видит, что что-то изменилось, автоматически перезапускает приложение.

Точнее, когда набор инструментов DevTools активен, приложение загружается в два отдельных загрузчика классов в виртуальной машине Java (JVM). Один загружает ваш код на Java, файлы свойств и почти все, что находится в пути `src/main/` проекта. Эти элементы могут часто меняться. Другой загрузчик загружает библиотеки зависимостей, которые вряд ли будут меняться так часто.

Обнаружив изменения, DevTools перезапускает только загрузчик классов, загружающий код вашего проекта, и повторно инициализирует контекст приложения Spring, но оставляет другой загрузчик классов и JVM нетронутыми. Эта стратегия позволяет немного сократить время, необходимое для запуска приложения.

Недостаток данной стратегии – изменения в зависимостях не будут вызывать автоматический перезапуск. Это связано с тем, что загрузчик классов, управляющий библиотеками зависимостей, не перезагружается автоматически. Каждый раз, добавляя, изменяя или удаляя зависимость в своей спецификации сборки, вы должны явно выполнить перезапуск приложения, чтобы эти изменения вступили в силу.

Автоматическое обновление браузера и отключение кеширования шаблонов

По умолчанию результаты анализа шаблонов, таких как Thymeleaf и FreeMarker, кешируются, поэтому шаблоны не нужно повторно анализировать при обслуживании каждого запроса. Это отлично подходит для промышленных условий, потому что дает небольшое преимущество в производительности.

Кеширование, однако, плохо подходит для этапа разработки, так как не позволяет оперативно видеть результаты внесения изменений в шаблоны после обновления окна браузера. Даже если вы внесли изменения, приложение продолжит использовать кешированный шаблон, пока вы не перезапустите его.

DevTools решает эту проблему, автоматически отключая кеширование всех шаблонов. Вносите в свои шаблоны столько изменений, сколько хотите, и знайте, что для просмотра изменений достаточно лишь обновить браузер.

Но если вы так же ленивы, как я, то наверняка не захотите утруждать себя даже такой малостью, как щелчок на кнопке обновления браузера. Было бы намного удобнее, если бы можно было вносить изменения и тут же наблюдать результаты в браузере. К счастью, в DevTools есть что предложить тем из нас, кому лень нажимать кнопку обновления.

DevTools автоматически запускает сервер LiveReload (<http://livere-load.com/>) вместе с приложением. Сам по себе сервер LiveReload не особенно полезен. Но в сочетании с соответствующим плагином LiveReload ваш браузер автоматически будет обновлять страницу при внесении изменений в шаблоны, изображения, таблицы стилей, сценарии JavaScript и т. д. – почти все, что в конечном итоге передается браузеру.

Плагины LiveReload имеются для браузеров Google Chrome, Safari и Firefox. (Извините, любители Internet Explorer и Edge.) Посетите страницу <http://livereload.com/extensions/>, чтобы найти информацию о том, как установить плагин LiveReload в свой браузер.

Встроенная консоль H2

Наш проект пока не использует базу данных, но это случится в главе 3. Если вы решите использовать базу данных H2 для разработки, то DevTools также автоматически запустит консоль H2, к которой можно получить доступ из веб-браузера. Достаточно ввести в браузере адрес `http://localhost:8080/h2-console`, и вы увидите данные, с которыми работает ваше приложение.

На данный момент мы написали законченное, хотя и очень простое приложение Spring. Мы будем расширять и дополнять его на протяжении всей книги. А сейчас самое время отступить на шаг назад и проанализировать, чего мы достигли и какую роль во всем этом сыграл фреймворк Spring.

1.3.6 Обзор результатов

Давайте перечислим, что мы уже сделали. Проще говоря, перечислим шаги, выполненные для создания приложения Taco Cloud:

- создали начальную структуру проекта с помощью Spring Initializr;
- написали класс контроллера для обработки запроса к домашней странице;
- определили шаблон представления для отображения домашней страницы;
- написали простой тестовый класс, проверяющий работу контроллера домашней страницы.

Как будто ничего сложного, верно? За исключением первого шага – создания проекта, – все последующие действия были нацелены на создание домашней страницы.

Фактически почти каждая строка написанного нами кода нацелена на это. Если не считать операторов импорта, то получается, что мы написали только две строки кода в классе контроллера и ни одной строки в шаблоне представления. И хотя большая часть тестового класса использует поддержку тестирования Spring, в контексте теста это почти незаметно.

Это важное преимущество разработки с использованием Spring. Вы можете сосредоточиться на прикладной логике, а не на удовлетворении требований фреймворка. Конечно, иногда приходится писать код, специфичный для фреймворка, но обычно этот код составляет лишь малую часть кодовой базы. Как я уже говорил, Spring (и Spring Boot) можно считать *фреймворком без фреймворка*.

Как такое вообще возможно? Что делает Spring за кулисами, чтобы гарантировать удовлетворение потребностей вашего приложения?

Давайте начнем расследование с того, что делает Spring, с обзора спецификации сборки.

В файле `pom.xml` мы объявили зависимости `spring-boot-starter-web` и `spring-boot-starter-thymeleaf`. Они транзитивно подключают ряд других зависимостей, в том числе:

- фреймворк Spring MVC;
- встроенный сервер Tomcat;
- механизм шаблонов Thymeleaf иialect языка разметки Thymeleaf.

Также мы автоматически получили библиотеку автоконфигурации Spring Boot. Когда приложение запускается, механизм автоконфигурации Spring Boot обнаруживает указанные нами зависимости и автоматически выполняет следующие шаги:

- настраивает bean-компоненты в контексте приложения Spring для включения Spring MVC;
- настраивает встроенный сервер Tomcat в контексте приложения Spring;
- настраивает механизм шаблонов Thymeleaf для отображения представлений Spring MVC.

Проще говоря, автоконфигурация выполняет всю рутинную работу, позволяя нам сосредоточиться на прикладном коде, реализующем функциональность приложения. Довольно удобное соглашение, как по мне!

Ваше путешествие по Spring только началось. В приложении Taco Cloud мы затронули лишь малую часть того, что может предложить Spring. Прежде чем сделать следующий шаг, давайте оглядимся вокруг и посмотрим, какие достопримечательности в ландшафте Spring ожидают нас в нашем путешествии.

1.4 Обзор ландшафта Spring

Чтобы получить представление о ландшафте Spring, обратите внимание на огромный список флагков в полной версии веб-формы Spring Initializr. В нем более 100 видов зависимостей, поэтому я даже не буду пытаться перечислить их здесь или предоставить скриншот – посмотрите сами, – но отмечу некоторые основные моменты.

1.4.1 Ядро Spring Framework

Как нетрудно догадаться, ядро Spring Framework является основой для всего остального во вселенной Spring. Это базовый контейнер и инфраструктура внедрения зависимостей. Но оно также предоставляет несколько других важных механизмов.

Среди них Spring MVC – веб-фреймворк Spring. Вы уже пробовали использовать Spring MVC для создания класса контроллера, обрабаты-

вающего веб-запросы. Но вы еще не видели, что Spring MVC можно использовать для создания REST API, генерирующего данные, отличные от HTML. Более детально мы изучим Spring MVC в главе 2, а затем еще раз посмотрим, как использовать его для создания REST API, в главе 7.

Ядро Spring Framework также предлагает некоторую элементарную поддержку хранения данных, в частности поддержку JDBC на основе шаблонов. Так, например, в главе 3 мы познакомимся с `JdbcTemplate`.

Spring включает поддержку реактивного программирования, в том числе новую реактивную веб-инфраструктуру Spring WebFlux, которая в значительной степени основана на Spring MVC. С моделью реактивного программирования в Spring мы познакомимся в части III книги и с Spring WebFlux в главе 12.

1.4.2 Spring Boot

Мы уже видели многие преимущества Spring Boot, включая подключение начальных зависимостей и автоконфигурацию. В этой книге мы используем Spring Boot в максимально возможной степени, чтобы избежать использования любых форм явной настройки, за исключением ситуаций, когда это абсолютно необходимо. В дополнение к начальным зависимостям и автонастройке Spring Boot предлагает также другие полезные возможности:

- Actuator позволяет исследовать работу внутренних механизмов приложения во время выполнения, включая получение метрик, дампов потоков выполнения и приложения и свойства окружения приложения;
- гибкую спецификацию свойств окружения;
- дополнительную поддержку тестирования, помимо поддержки, предоставляемой ядром фреймворка.

Кроме того, Spring Boot предлагает альтернативную модель программирования, основанную на сценариях Groovy, которая называется Spring Boot CLI (Command Line Interface – интерфейс командной строки). С помощью Spring Boot CLI можно писать целые приложения в виде набора сценариев Groovy и запускать их из командной строки. Мы не будем тратить много времени на Spring Boot CLI, но будем обращаться к нему при случае, когда это будет соответствовать нашим потребностям.

Spring Boot стал настолько неотъемлемой частью разработки с использованием Spring, что я не могу представить разработку Spring-приложений без него. Поэтому данная книга в равной степени ориентирована и на Spring Boot, и вы часто будете замечать, что я использую слово Spring, когда имею в виду то, что делает Spring Boot.

1.4.3 Spring Data

Ядро Spring Framework уже включает базовую поддержку хранения данных, но Spring Data – это нечто совершенно удивительное! Этот

компонент экосистемы Spring позволяет определять репозитории данных для приложений в форме простых интерфейсов Java, используя соглашение об именах при определении методов, реализующих хранение и извлечение данных.

Более того, Spring Data может работать с несколькими различными типами баз данных, включая реляционные (через JDBC или JPA), документные (Mongo), графовые (Neo4j) и др. Мы будем использовать Spring Data для создания репозиториев для приложения Taco Cloud в главе 3.

1.4.4 *Spring Security*

Безопасность приложений всегда была важной темой, и кажется, что с каждым днем ее важность только возрастает. К счастью, в Spring есть надежный фреймворк Spring Security.

Spring Security удовлетворяет широкий спектр потребностей в обеспечении безопасности приложений, включая аутентификацию, авторизацию и защиту API. Спектр возможностей Spring Security слишком велик, чтобы его можно было должным образом охватить в этой книге, но все же мы коснемся некоторых наиболее распространенных вариантов его применения в главах 5 и 12.

1.4.5 *Spring Integration и Spring Batch*

Многим приложениям в какой-то момент требуется интегрироваться с другими приложениями или даже с другими компонентами того же приложения. Для удовлетворения этих потребностей появилось несколько моделей интеграции приложений. Spring Integration и Spring Batch обеспечивают реализацию этих моделей для приложений Spring.

Spring Integration обеспечивает интеграцию в реальном времени, когда данные обрабатываются по мере их поступления. Spring Batch, напротив, предназначен для пакетной интеграции, когда данные могут собираться в течение некоторого времени, пока не произойдет какое-то событие (например, сработает таймер), сообщающее о том, что пришла пора для обработки пакета данных. С фреймворком Spring Integration мы познакомимся в главе 10.

1.4.6 *Spring Cloud*

Мир разработки приложений вступает в новую эру, когда мы перестанем разрабатывать наши приложения как единые монолиты и будем составлять их из множества отдельных единиц развертывания, известных как **микросервисы**.

Микросервисы – актуальное направление, решающее множество практических проблем разработки и выполнения. Но при этом они привносят свои собственные проблемы. Эти проблемы решает Spring

Cloud – набор проектов для разработки облачных приложений с помощью Spring.

Spring Cloud охватывает множество направлений, и было бы невозможно осветить их все в этой книге. Если вас заинтересует данная тема, то я советую прочитать книгу «Cloud Native Spring in Action» Томаса Витале (Thomas Vitale; Manning, 2020, www.manning.com/books/cloud-native-spring-in-action).

1.4.7 *Spring Native*

Проект Spring Native является относительно новой экспериментальной разработкой в экосистеме Spring. Он позволяет компилировать проекты Spring Boot в двоичные выполняемые файлы с помощью компилятора GraalVM. Такие файлы запускаются значительно быстрее и занимают меньше места.

За дополнительной информацией о Spring Native обращайтесь по адресу <https://github.com/spring-projects-experimental/spring-native>.

Итоги

- Spring стремится упростить решение таких задач, как создание веб-приложений, работа с базами данных, защита приложений и микросервисы.
- Spring Boot основан на Spring и делает работу с фреймворком Spring еще проще, предлагая автоматическое управление зависимостями и конфигурацией и возможность анализа параметров выполнения приложений.
- Приложения Spring можно инициализировать с помощью веб-приложения Spring Initializr, доступного в интернете, которое поддерживается большинством сред разработки на Java.
- Компоненты, обычно называемые bean-компонентами, в контексте приложения Spring могут объявляться явно с помощью Java или XML, обнаруживаться путем сканирования компонентов или автоматически настраиваться с помощью механизма автоконфигурации в Spring Boot.

Разработка веб-приложений

В этой главе рассматриваются следующие темы:

- модель представления данных в браузере;
- проверка и обработка ввода;
- выбор библиотеки шаблонов представлений.

Первые впечатления – самые важные. Внешний вид дома может вызвать желание купить его еще до того, как покупатель войдет внутрь. Вишневый цвет автомобиля привлекает больше внимания, чем то, что находится под капотом. А литература изобилует историями о любви с первого взгляда. Безусловно, то, что внутри, важно, но не менее важно и то, что снаружи, – то, что видно сразу.

Приложения, которые вы будете создавать с помощью Spring, могут выполнять множество нужных и полезных действий: обрабатывать данные, читать информацию из базы данных и взаимодействовать с другими приложениями. Но первое впечатление, которое получат пользователи от вашего приложения, будет связано с пользовательским интерфейсом. И во многих приложениях пользовательский интерфейс оформлен в виде веб-страниц, отображаемых в браузере.

В главе 1 мы создали свой первый контроллер Spring MVC для отображения домашней страницы нашего приложения. Но Spring MVC способен не только отображать статический контент. В этой главе мы разработаем первую крупную функцию приложения Taco Cloud – возможность составлять рецепт тако из ингредиентов. Попутно мы углубимся в Spring MVC и научимся отображать модели данных и обрабатывать формы.

2.1 Отображение информации

По сути, Taco Cloud – это приложение, с помощью которого можно заказать тако через интернет. Однако владельцы Taco Cloud хотели бы дать своим клиентам возможность выразить свои творческие способности и составить свой рецепт тако из богатой палитры ингредиентов.

Поэтому веб-приложению Taco Cloud нужна страница, на которой бы отображался выбор ингредиентов для мастеров составления рецептов тако. Выбор ингредиентов может измениться в любое время, поэтому они не должны быть жестко «зашиты» в код HTML страницы. Приложение должно извлекать список доступных ингредиентов из базы данных и передавать в страницу для отображения.

Извлечением и обработкой данных в веб-приложениях Spring занимаются контроллеры. А преобразованием этих данных в разметку HTML, которая будет отображаться в браузере, занимаются представления. Соответственно, для поддержки страницы мы создадим следующие компоненты:

- класс, определяющий свойства ингредиентов тако;
- класс контроллера Spring MVC, извлекающий информацию об ингредиентах и передающий ее в представление;
- шаблон представления, отображающий список ингредиентов в браузере.

Взаимосвязи между этими компонентами показаны на рис. 2.1.

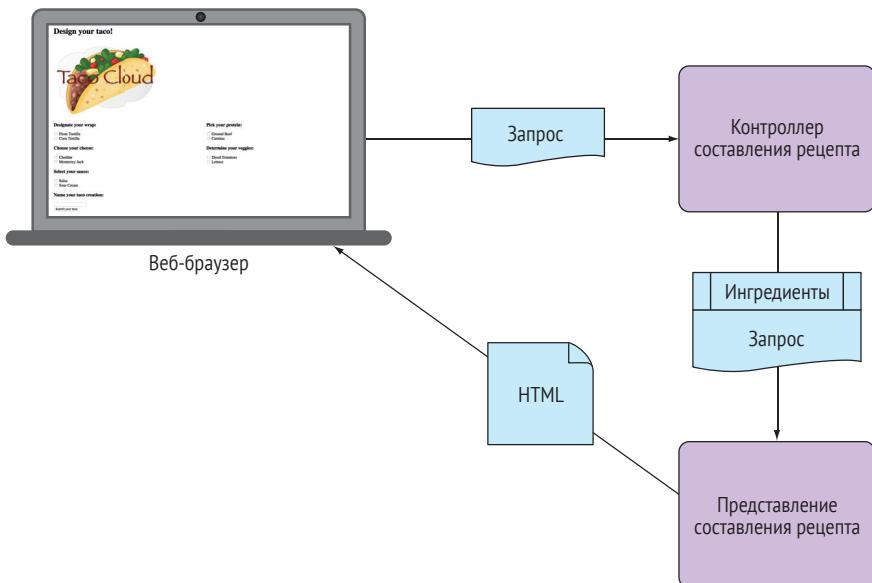


Рис. 2.1 Типичный поток обработки данных с помощью Spring MVC

В этой главе основное внимание будет сконцентрировано на веб-фреймворке Spring, поэтому мы отложим все, что связано с базой данных, до главы 3. Пока всю ответственность за передачу ингредиентов в представление мы возложим на контроллер. Но в главе 3 мы переделаем контроллер так, чтобы он взаимодействовал с репозиторием, который будет извлекать ингредиенты из базы данных.

Прежде чем приступить к созданию контроллера и представления, давайте поработаем над предметной областью ингредиентов. Так мы получим основу, на которой сможем строить свои веб-компоненты.

2.1.1 Предметная область ингредиентов

Предметная область приложения – это идеи и концепции, влияющие на понимание приложения¹. В приложении Taco Cloud предметная область включает такие объекты, рецепты тако, ингредиенты, из которых состоят эти рецепты, клиенты и заказы, размещенные клиентами. На рис. 2.2 показаны эти объекты и как они взаимосвязаны.

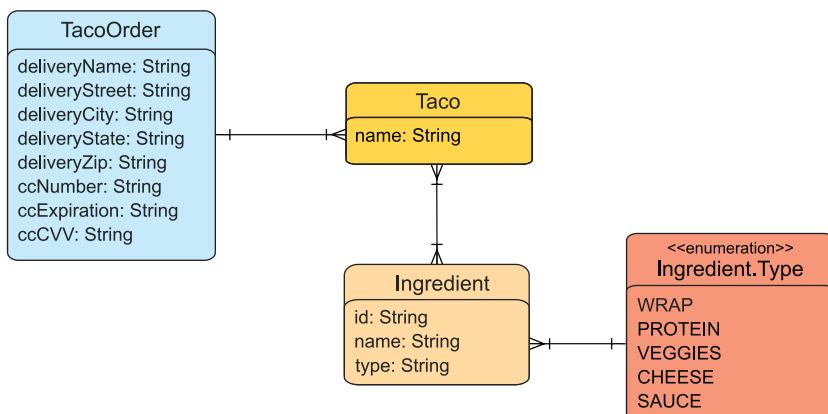


Рис. 2.2 Предметная область Taco Cloud

Для начала сосредоточимся на ингредиентах тако. В нашей предметной области ингредиенты для тако – довольно простые объекты. У каждого ингредиента есть название и тип, позволяющие его визуально классифицировать (белки, сыры, соусы и т. д.). У каждого также есть идентификатор, по которому на него можно ссылаться. Следующий класс `Ingredient` определяет соответствующий объект предметной области (листинг 2.1).

¹ Более детальное обсуждение понятия предметной области приложений вы найдете в книге Эрика Эванса (Eric Evans) «Domain-Driven Design», выпущенной издательством Addison-Wesley Professional в 2003 году. (Эванс Эрик. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. Вильямс, 2020, ISBN: 978-5-6040724-9-3. – Прим. перев.)

Листинг 2.1 Класс, представляющий ингредиенты тако

```
package tacos;

import lombok.Data;

@Data
public class Ingredient {

    private final String id;
    private final String name;
    private final Type type;

    public enum Type {
        WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
    }

}
```

Как видите, это самый заурядный класс данных на Java, определяющий три свойства, описывающих ингредиент. Возможно, самое необычное в классе `Ingredient` – отсутствие обычного набора методов чтения и записи (геттеров и сеттеров), не говоря уже о таких полезных методах, как `equals()`, `hashCode()`, `toString()` и др.

Отчасти их отсутствие объясняется экономией места, а отчасти – использованием замечательной библиотеки Lombok, которая автоматически генерирует эти методы во время компиляции. На самом деле аннотация `@Data` на уровне класса реализована в Lombok и сообщает этой библиотеке, что она должна генерировать все эти отсутствующие методы, а также конструктор, который принимает значения свойств в аргументах. Использование Lombok позволило ограничиться коротким и аккуратным определением класса `Ingredient`.

Lombok не входит в состав Spring, но она настолько полезна, что мне трудно представить разработку без нее. Кроме того, она выручает меня, когда мне нужно сделать примеры кода в книге короткими и понятными.

Чтобы использовать Lombok, ее нужно добавить в зависимости проекта. Если вы используете Spring Tool Suite, то для этого достаточно щелкнуть правой кнопкой мыши на файле `pom.xml` и выбрать в контекстном меню пункт **Add Starters** (Добавить начальные зависимости). В ответ появится тот же набор зависимостей, что был показан в главе 1 (рис. 1.4), где вы сможете добавить или удалить зависимости. Найдите зависимость Lombok в разделе **Developer Tools** (Инструменты разработчика), выберите ее и щелкните на кнопке **OK**; после этого Spring Tool Suite автоматически добавит библиотеку в спецификацию сборки.

То же самое можно сделать вручную, добавив следующие строки в `pom.xml`:

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
```

Если вы решите добавить Lombok вручную, то исключите ее в настройках плагина Spring Boot Maven в разделе `<build>` в файле `pom.xml`:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <configuration>
        <excludes>
          <exclude>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
          </exclude>
        </excludes>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Свое волшебство Lombok творит во время компиляции, поэтому она не нужна во время выполнения. Исключение ее, как показано выше, не позволит включить ее в окончательный файл JAR или WAR.

Добавив зависимость Lombok, вы получаете доступ к аннотациям Lombok (например, `@Data`) во время разработки и автоматическое создание методов во время компиляции. Но вам также нужно добавить Lombok как расширение в вашу среду разработки (IDE), иначе она будет сообщать об ошибках отсутствия методов и финальных (`final`) свойств без сеттеров. Инструкции по установке Lombok в вашу IDE вы найдете по адресу <https://projectlombok.org/>.

Почему в моем коде так много ошибок?

Стоит повторить еще раз, что при использовании Lombok необходимо установить плагин Lombok в среду разработки. Без этого она не будет знать, что геттеры, сеттеры и другие методы будут генерироваться автоматически библиотекой Lombok, и будет реагировать на их отсутствие в коде как на ошибки.

Lombok поддерживается многими популярными IDE, включая Eclipse, Spring Tool Suite, IntelliJ IDEA и Visual Studio Code. Дополнительную информацию об установке плагина Lombok в вашу среду разработки вы найдете по адресу <https://projectlombok.org/>.

Я думаю, вам понравится библиотека Lombok, но знайте, что пользоваться ею совсем необязательно. Она не требуется для разработки приложений Spring, поэтому если вы не хотите ее использовать, то просто пишите недостающие методы вручную. Я никуда не спешу и готов подождать вас.

Ингредиенты – это основные строительные блоки тако. Чтобы понять, как они объединяются, мы определим класс Taco (листинг 2.2).

Листинг 2.2 Класс, представляющий рецепт

```
package tacos;
import java.util.List;
import lombok.Data;

@Data
public class Taco {

    private String name;
    private List<Ingredient> ingredients;

}
```

Как видите, Taco – это еще один обычный класс данных на Java с парой свойств. Как и Ingredient, класс Taco снабжен аннотацией @Data, чтобы Lombok автоматически сгенерировала необходимые методы JavaBean во время компиляции.

Теперь, когда у нас есть классы Ingredient и Taco, определим еще один класс данных, TacoOrder, представляющий заказы, которые клиенты будут оставлять на сайте, с информацией о рецепте, оплате и доставке (листинг 2.3).

Листинг 2.3 Класс, представляющий заказ

```
package tacos;
import java.util.List;
import java.util.ArrayList;
import lombok.Data;

@Data
public class TacoOrder {

    private String deliveryName;
    private String deliveryStreet;
    private String deliveryCity;
    private String deliveryState;
    private String deliveryZip;
    private String ccNumber;
    private String ccExpiration;
    private String ccCVV;

    private List<Taco> tacos = new ArrayList<>();

    public void addTaco(Taco taco) {
        this.tacos.add(taco);
    }
}
```

Кроме того что он имеет намного больше свойств, чем `Ingredient` или `Taco`, в классе `TacoOrder` нет ничего нового для обсуждения. Это простой класс данных с девятью свойствами: пять содержат информацию о доставке, три – об оплате и одно – список объектов `Taco`, составляющих заказ. Также класс имеет метод `addTaco()`, добавленный для удобства добавления тако в заказ.

Теперь, когда предметная область определена, можно приступать к основной работе. Давайте добавим в приложение несколько контроллеров для обработки веб-запросов.

2.1.2 Создание класса контроллера

Контроллеры – это основные игроки в Spring MVC. Их основная задача – обрабатывать HTTP-запросы и либо передавать результаты в представление для преобразования в HTML (и отображения в браузере), либо записывать их непосредственно в тело ответа (RESTful). В этой главе мы рассмотрим контроллеры, которые передают результаты представлениям, генерирующими контент для веб-браузеров. А контроллерами, обрабатывающими запросы к REST API, мы займемся в главе 7.

Для приложения `Taco Cloud` нам понадобится простой контроллер, выполняющий следующие действия:

- принимает и обрабатывает HTTP-запросы `GET` с путем `/design`;
- составляет список ингредиентов;
- передает запрос и ингредиенты в шаблон представления, который будет преобразован в HTML и отправлен веб-браузеру.

Класс `DesignTacoController` этого контроллера показан в листинге 2.4.

Листинг 2.4 Класс первого контроллера

```
package tacos.web;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttributes;

import lombok.extern.slf4j.Slf4j;
import tacos.Ingredient;
import tacos.Ingredient.Type;
import tacos.Taco;
```

```
@Slf4j
@Controller
@RequestMapping("/design")
@SessionAttributes("tacoOrder")
public class DesignTacoController {

    @ModelAttribute
    public void addIngredientsToModel(Model model) {
        List<Ingredient> ingredients = Arrays.asList(
            new Ingredient("FLTO", "Flour Tortilla", Type.WRAP),
            new Ingredient("COTO", "Corn Tortilla", Type.WRAP),
            new Ingredient("GRBF", "Ground Beef", Type.PROTEIN),
            new Ingredient("CARN", "Carnitas", Type.PROTEIN),
            new Ingredient("TMTO", "Diced Tomatoes", Type.VEGGIES),
            new Ingredient("LETC", "Lettuce", Type.VEGGIES),
            new Ingredient("CHED", "Cheddar", Type.CHEESE),
            new Ingredient("JACK", "Monterrey Jack", Type.CHEESE),
            new Ingredient("SLSA", "Salsa", Type.SAUCE),
            new Ingredient("SRCR", "Sour Cream", Type.SAUCE)
        );
        Type[] types = Ingredient.Type.values();
        for (Type type : types) {
            model.addAttribute(type.toString().toLowerCase(),
                filterByType(ingredients, type));
        }
    }

    @ModelAttribute(name = "tacoOrder")
    public TacoOrder order() {
        return new TacoOrder();
    }

    @ModelAttribute(name = "taco")
    public Taco taco() {
        return new Taco();
    }

    @GetMapping
    public String showDesignForm() {
        return "design";
    }

    private Iterable<Ingredient> filterByType(
        List<Ingredient> ingredients, Type type) {
        return ingredients
            .stream()
            .filter(x -> x.getType().equals(type))
            .collect(Collectors.toList());
    }
}
```

Первое, что следует отметить в `DesignTacoController`, – это набор аннотаций, применяемых на уровне класса. Аннотация `@Slf4j` реализована в Lombok. Во время компиляции она автоматически генерирует статическое свойство типа `Logger`, определяемого библиотекой SLF4J (Simple Logging Facade for Java – простой интерфейс журналирования для Java, <https://www.slf4j.org/>). Эта скромная аннотация за кулисами добавляет в класс следующее определение:

```
private static final org.slf4j.Logger log =
    org.slf4j.LoggerFactory.getLogger(DesignTacoController.class);
```

Мы воспользуемся этим свойством чуть ниже.

Следующая аннотация, применяемая к `DesignTacoController`, – `@Controller`. Она идентифицирует этот класс как контроллер и отмечает его как доступный для механизма сканирования компонентов, чтобы фреймворк Spring мог обнаружить его и автоматически создать экземпляр `DesignTacoController` в виде bean-компонента в контексте приложения.

`DesignTacoController` также снабжен аннотацией `@RequestMapping`. При применении на уровне класса аннотация `@RequestMapping` определяет тип запросов, которые обрабатывает этот контроллер. В данном случае она сообщает, что `DesignTacoController` будет обрабатывать запросы, пути в которых начинаются с `/design`.

Наконец, класс `DesignTacoController` снабжен аннотацией `@SessionAttributes("tacoOrder")`. Она указывает, что объект `TacoOrder`, объявленный в классе чуть ниже, должен поддерживаться на уровне сеанса. Это важно, потому что создание тако также является первым шагом в создании заказа, и созданный нами заказ необходимо будет перенести в сеанс, охватывающий несколько запросов.

ОБРАБОТКА ЗАПРОСА GET

Аннотация `@RequestMapping` на уровне класса дополнена аннотацией `@GetMapping` на уровне метода `showDesignForm()`. Аннотация `@GetMapping` в сочетании с `@RequestMapping` на уровне класса определяет метод, в данном случае `showDesignForm()`, который должен вызываться для обработки HTTP-запроса GET с путем `/design`.

`@GetMapping` – это всего лишь одна из целого семейства аннотаций сопоставления запросов. В табл. 2.1 перечислены все аннотации из этого семейства, доступные в Spring MVC.

Таблица 2.1 Аннотации сопоставления запросов в Spring MVC

Аннотация	Описание
<code>@RequestMapping</code>	Обобщенная обработка запросов
<code>@GetMapping</code>	Обработка HTTP-запросов GET
<code>@PostMapping</code>	Обработка HTTP-запросов POST
<code>@PutMapping</code>	Обработка HTTP-запросов PUT
<code>@DeleteMapping</code>	Обработка HTTP-запросов DELETE
<code>@PatchMapping</code>	Обработка HTTP-запросов PATCH

Метод `showDesignForm()`, обрабатывающий запросы GET с путем `/design`, на самом деле почти ничего не делает. Он просто возвращает строковое значение `"design"` – логическое имя представления, которое будет использоваться для отображения модели в браузере. Но перед этим он добавляет в модель `Model` пустой объект `Taco` с ключом `"design"`, который послужит чистым холстом, на котором клиент сможет изобразить свой шедевр тако.

Казалось бы, GET-запрос с путем `/design` почти ничего не делает. Но на самом деле он приводит в движение многие механизмы, невидимые за вызовом метода `showDesignForm()`. Обратите также внимание на метод `addIngredientsToModel()`, который снабжен аннотацией `@ModelAttribute`. Этот метод тоже будет вызываться в процессе обработки запроса и создавать список объектов `Ingredient`, который затем будет помещен в модель. Пока список жестко «зашит» в код. Но в главе 3 мы реализуем его получение из базы данных.

После подготовки списка ингредиентов следующие несколько строк в `addIngredientsToModel()` фильтруют ингредиенты по типам, используя вспомогательный метод `filterByType()`. Затем список типов ингредиентов добавляется в виде атрибута в объект модели `Model`, который будет передан в вызов `showDesignForm()`. `Model` – это объект, в котором данные пересыпаются между контроллером и любым представлением, ответственным за преобразование этих данных в разметку HTML. В конечном итоге данные, помещенные в атрибуты модели, копируются в атрибуты запроса сервлета, где представление найдет их и использует для отображения страницы в браузере пользователя.

За `addIngredientsToModel()` следуют еще два метода, также снабженных аннотацией `@ModelAttribute`. Эти методы намного проще и просто создают новые объекты `TacoOrder` и `Taco` для размещения в модели. Объект `TacoOrder`, упомянутый выше в аннотации `@SessionAttributes`, хранит состояние собираемого заказа, пока клиент выбирает ингредиенты для тако несколькими запросами. Объект `Taco` помещается в модель, чтобы представление, отображаемое в ответ на запрос GET с путем `/design`, имело объект для отображения.

Наш контроллер `DesignTacoController` начинает обретать форму. Если сейчас запустить приложение и ввести в браузере путь `/design`, то приложение вызвало бы методы `showDesignForm()` и `addIngredientsToModel()` контроллера `DesignTacoController`, помещающие список ингредиентов и пустой объект `Taco` в модель перед передачей запроса в представление. Но мы пока не определили представление, поэтому такая попытка приведет к ошибке HTTP 500 (внутренняя ошибка сервера). Чтобы не допустить этого, переключим наше внимание на представление, в котором данные будут включены в разметку HTML для отображения в веб-браузере пользователя.

2.1.3 Создание представления

После того как контроллер закончит свою работу, в игру вступает представление. Spring предлагает несколько отличных вариантов

определения представлений, включая JavaServer Pages (JSP), Thymeleaf, FreeMarker, Mustache и шаблоны Groovy. Мы будем использовать Thymeleaf, как было решено в главе 1, когда мы создавали проект. Ниже, в разделе 2.5, рассмотрим несколько других вариантов.

Мы уже добавили Thymeleaf в проект как зависимость. На этапе автокофигурация Spring Boot видит, что Thymeleaf находится в пути поиска классов classpath, и автоматически создает bean-компоненты, поддерживающие представления Thymeleaf.

Библиотеки представлений, такие как Thymeleaf, предназначены для отделения представления данных от конкретного веб-фреймворка. В результате эти библиотеки ничего не знают об абстракции модели и не могут работать с данными, которые контроллер помещает в модель. Но они могут работать с атрибутами запроса сервера. Поэтому, перед тем как Spring передаст запрос представлению, данные модели копируются в атрибуты запроса, которые могут использовать Thymeleaf и другие механизмы шаблонов представлений.

Шаблоны Thymeleaf – это самый обычный код HTML с некоторыми дополнительными атрибутами элементов, которые управляют преобразованием данных в механизме шаблонов. Например, если есть атрибут запроса с ключом "message" и вы хотите, чтобы Thymeleaf преобразовал его в HTML-тег `<p>`, то вы должны написать следующее выражение на языке шаблонов Thymeleaf:

```
<p th:text="${message}">placeholder message</p>
```

Когда шаблон преобразуется в HTML, тело элемента `<p>` будет заменено значением атрибута запроса сервера с ключом "message". Атрибут `th:text` – это атрибут пространства имен Thymeleaf, который выполняет замену. Оператор `${}` определяет имя атрибута (в данном случае "message"), значение которого следует использовать.

Thymeleaf также предлагает еще один атрибут, `th:each`, который выполняет обход коллекции элементов и каждый из них преобразует в код HTML. Этот атрибут удобно использовать для вывода списка ингредиентов. Например, чтобы отобразить только список "шаг" ингредиентов, можно использовать такой фрагмент на языке шаблонов:

```
<h3>Designate your шаг:</h3>
<div th:each="ingredient : ${шаг}">
    <input th:field="*{ingredients}" type="checkbox"
        th:value="${ingredient.id}"/>
    <span th:text="${ingredient.name}">INGREDIENT</span><br/>
</div>
```

Здесь атрибут `th:each` в теге `<div>` создает теги `<div>` снова и снова для каждого элемента в коллекции, указанной в атрибуте запроса шаг. В каждой итерации очередной элемент списка присваивается переменной с именем `ingredient`.

Внутри элемента `<div>` находится элемент флажка `<input>` и элемент ``, определяющий подпись для флажка. Атрибут `th:value`

устанавливает значение атрибута `value` отображаемого элемента `<input>`, получая его из свойства `id` ингредиента. Наконец, атрибут `th:field` устанавливает атрибут `name` элемента `<input>` и его состояние – отмечен / не отмечен. Когда позже мы добавим проверку, это гарантирует, что флагок сохранит свое состояние, если форму понадобится повторно отобразить после ошибки проверки. Элемент `` использует атрибут `th:text` для замены текста-заполнителя "INGREDIENT" значением свойства `name` ингредиента.

При преобразовании реальных данных модели одна итерация этого цикла `<div>` может выглядеть так:

```
<div>
    <input name="ingredients" type="checkbox" value="FLTO" />
    <span>Flour Tortilla</span><br/>
</div>
```

Предыдущий фрагмент на языке шаблонов Thymeleaf является лишь частью более крупной HTML-формы, с помощью которой наши клиенты будут творить свои кулинарные шедевры. Полный шаблон Thymeleaf, включая все типы ингредиентов и форму, показан в листинге 2.5.

Листинг 2.5 Страница составления рецепта тако

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Taco Cloud</title>
    <link rel="stylesheet" th:href="@{/styles.css}" />
</head>

<body>
    <h1>Design your taco!</h1>
    

    <form method="POST" th:object="${taco}">
        <div class="grid">
            <div class="ingredient-group" id="wraps">
                <h3>Designate your wrap:</h3>
                <div th:each="ingredient : ${wrap}">
                    <input th:field="*{ingredients}" type="checkbox"
                           th:value="${ingredient.id}"/>
                    <span th:text="${ingredient.name}">INGREDIENT</span><br/>
                </div>
            </div>

            <div class="ingredient-group" id="proteins">
                <h3>Pick your protein:</h3>
                <div th:each="ingredient : ${protein}">
                    <input th:field="*{ingredients}" type="checkbox"
                           th:value="${ingredient.id}"/>
                </div>
            </div>
        </div>
    </form>
</body>
```

```

        <span th:text="${ingredient.name}">INGREDIENT</span><br />
    </div>
</div>

<div class="ingredient-group" id="cheeses">
    <h3>Choose your cheese:</h3>
    <div th:each="ingredient : ${cheese}">
        <input th:field="*{ingredients}" type="checkbox"
               th:value="${ingredient.id}" />
        <span th:text="${ingredient.name}">INGREDIENT</span><br />
    </div>
</div>

<div class="ingredient-group" id="veggies">
    <h3>Determine your veggies:</h3>
    <div th:each="ingredient : ${veggies}">
        <input th:field="*{ingredients}" type="checkbox"
               th:value="${ingredient.id}" />
        <span th:text="${ingredient.name}">INGREDIENT</span><br />
    </div>
</div>

<div class="ingredient-group" id="sauces">
    <h3>Select your sauce:</h3>
    <div th:each="ingredient : ${sauce}">
        <input th:field="*{ingredients}" type="checkbox"
               th:value="${ingredient.id}" />
        <span th:text="${ingredient.name}">INGREDIENT</span><br />
    </div>
</div>
</div>

<div>
    <h3>Name your taco creation:</h3>
    <input type="text" th:field="*{name}" />
    <br />
    <button>Submit Your Taco</button>
</div>
</form>
</body>
</html>

```

Как видите, фрагмент `<div>` повторяется для каждого типа ингредиентов и включает кнопку **Submit** (Отправить) и поле, где пользователь может дать название своему творению.

Также стоит отметить, что полный шаблон включает изображение логотипа Taco Cloud и ссылку `<link>` на таблицу стилей¹. В обоих случаях для создания контекстно-зависимого пути к статическим арте-

¹ Содержимое таблицы стилей не имеет отношения к нашему обсуждению; она содержит только стиль оформления списка ингредиентов в виде двух столбцов.

фактам используется оператор @{}. Как вы узнали в главе 1, статический контент в приложениях Spring Boot обслуживается из каталога /static.

Теперь, когда контроллер и представление готовы, можете попробовать запустить приложение и полюбоваться на плоды своего труда. Запустить приложение Spring Boot можно множеством способов. В главе 1 я показал, как запустить приложение щелчком на кнопке Start (Пуск) на панели инструментов Spring Boot. После запуска приложения Taco Cloud тем или иным способом введите в адресной строке браузера адрес <http://localhost:8080/design>. В результате должна появиться страница, похожая на изображенную на рис. 2.3.

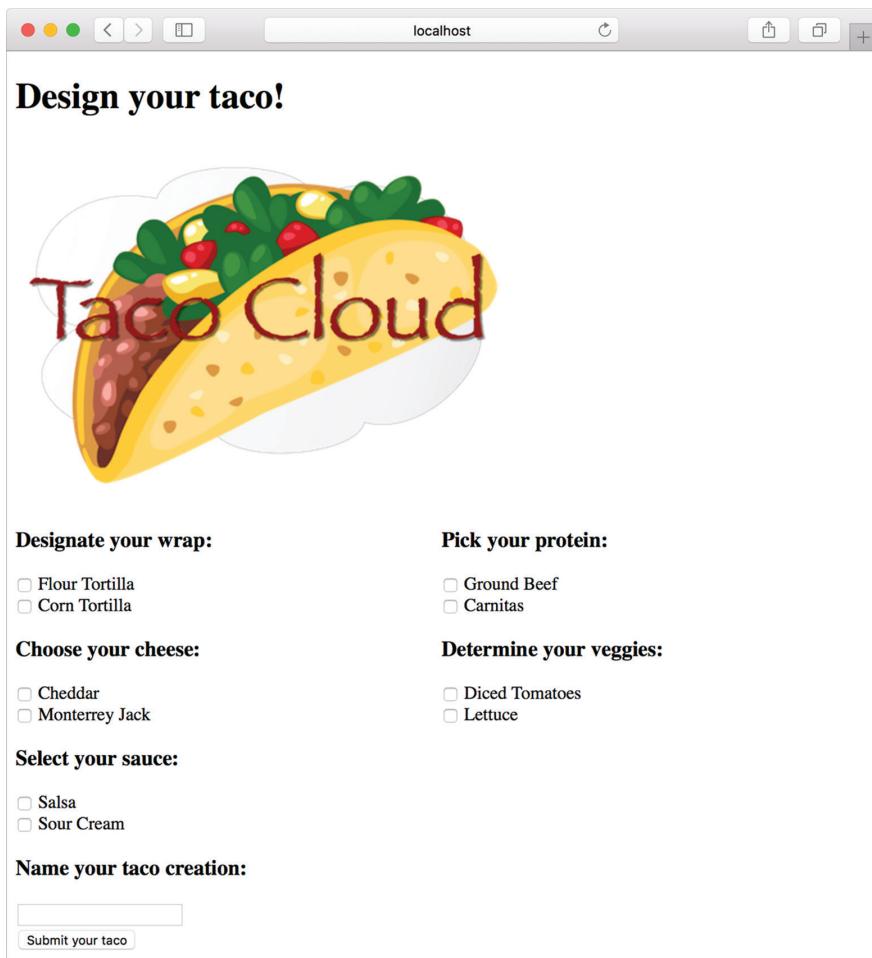


Рис. 2.3 Страница для составления рецепта тако

Выглядит довольно привлекательно! Мастеру тако, посетившему ваш сайт, предоставляется форма с палитрой ингредиентов для тако,

из которых он сможет создать свой шедевр. Но что происходит потом, когда он щелкнет на кнопке **Submit Your Taco** (Отправить рецепт тако)?

Наш контроллер `DesignTacoController` пока не готов принимать готовые творения. Если сейчас отправить форму, то вы получите сообщение об ошибке. (В данном случае это будет ошибка HTTP 405: метод запроса «POST» не поддерживается.) Давайте исправим эту проблему, добавив в контроллер еще немного кода, обрабатывающего отправленную форму.

2.2 Обработка отправленной формы

Взглянув еще раз на тег `<form>` в представлении, можно заметить, что атрибуту `method` присвоено значение `POST`. Более того, `<form>` не объявляет атрибут `action`. Это означает, что при отправке формы браузер собирает все данные в форме и отправит их на сервер в HTTP-запросе `POST` по тому же пути, по которому запрос `GET` получил форму, – по пути `/design`.

Поэтому нам нужно определить в контроллере метод, принимающий запросы `POST`. То есть нам нужно добавить новый метод в `DesignTacoController`, который бы обрабатывал запросы `POST` с путем `/design`.

В листинге 2.4 мы использовали аннотацию `@GetMapping`, чтобы указать, что метод `showDesignForm()` должен обрабатывать HTTP-запросы `GET` с путем `/design`. Аналогично мы можем использовать аннотацию `@PostMapping`, чтобы определить метод для обработки запросов `POST`. Давайте добавим в `DesignTacoController` метод `processTaco()`, как показано в листинге 2.6.

Листинг 2.6 Обработка запросов POST с использованием аннотации `@PostMapping`

```
@PostMapping
public String processTaco(Taco taco,
    @ModelAttribute TacoOrder tacoOrder) {
    tacoOrder.addTaco(taco);
    log.info("Processing taco: {}", taco);

    return "redirect:/orders/current";
}
```

Аннотация `@PostMapping`, которую мы применили к методу `processTaco()`, сообщает аннотации `@RequestMapping` на уровне класса, что `processTaco()` будет обрабатывать запросы `POST` с путем `/design`. Это именно то, что нам нужно для обработки творений мастеров тако.

При отправке поля формы присваиваются свойствам объекта `Taco` (соответствующий класс показан в листинге 2.2), который затем пере-

дается в качестве параметра методу `processTaco()`. Получив этот объект, метод `processTaco()` может делать с ним все, что захочет. В данном случае он добавляет объект Taco в объект `TacoOrder`, который также передается в параметре, а затем записывает его в журнал. Аннотация `@ModelAttribute` перед параметром `TacoOrder` указывает, что он должен использовать объект `TacoOrder`, который был помещен в модель методом `order()` с аннотацией `@ModelAttribute` (листинг 2.4).

Если вы снова посмотрите на форму в листинге 2.5, то увидите несколько элементов-флажков `checkbox` с именем `ingredients` и элемент поля ввода текста с именем `name`. Эти поля в форме напрямую соответствуют ингредиентам и свойствам имени класса `Taco`.

Поле `name` в форме должно содержать простое текстовое значение. То есть свойство `name` в объекте `Taco` имеет тип `String`. Флажки ингредиентов тоже имеют текстовые значения, но поскольку их может быть выбрано несколько или не выбран ни один, свойство `ingredients`, к которому они привязаны, имеет тип `List<Ingredient>` – список, содержащий все выбранные ингредиенты.

Но постойте-ка! Если флажки ингредиентов имеют текстовые значения, а объект `Taco` представляет список ингредиентов как `List<Ingredient>`, то не возникает ли здесь противоречия? Как список текстовых строк, такой как `["FLTO", "GRBF", "LETC"]`, можно присвоить списку объектов `Ingredient`, которые являются более сложными объектами, содержащими не только идентификатор, но также описательное имя и тип ингредиента?

Для решения этого противоречия нам пригодится конвертер. Конвертер – это любой класс, который реализует интерфейс `Converter` с методом `convert()`, получающим значение одного типа и преобразующим его в значение другого типа. Чтобы преобразовать `String` в `Ingredient`, мы будем использовать `IngredientByIdConverter`, как показано в листинге 2.7.

Листинг 2.7 Преобразование строк в объекты `Ingredient`

```
package tacos.web;

import java.util.HashMap;
import java.util.Map;
import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;

import tacos.Ingredient;
import tacos.Ingredient.Type;

@Component
public class IngredientByIdConverter implements Converter<String, Ingredient> {

    private Map<String, Ingredient> ingredientMap = new HashMap<>();

    public IngredientByIdConverter() {
        ingredientMap.put("FLTO",

```

```

        new Ingredient("FLTO", "Flour Tortilla", Type.WRAP));
ingredientMap.put("COTO",
        new Ingredient("COTO", "Corn Tortilla", Type.WRAP));
ingredientMap.put("GRBF",
        new Ingredient("GRBF", "Ground Beef", Type.PROTEIN));
ingredientMap.put("CARN",
        new Ingredient("CARN", "Carnitas", Type.PROTEIN));
ingredientMap.put("TMT0",
        new Ingredient("TMT0", "Diced Tomatoes", Type.VEGGIES));
ingredientMap.put("LETC",
        new Ingredient("LETC", "Lettuce", Type.VEGGIES));
ingredientMap.put("CHED",
        new Ingredient("CHED", "Cheddar", Type.CHEESE));
ingredientMap.put("JACK",
        new Ingredient("JACK", "Monterrey Jack", Type.CHEESE));
ingredientMap.put("SLSA",
        new Ingredient("SLSA", "Salsa", Type.SAUCE));
ingredientMap.put("SRCR",
        new Ingredient("SRCR", "Sour Cream", Type.SAUCE));
}

@Override
public Ingredient convert(String id) {
    return ingredientMap.get(id);
}
}

```

У нас пока нет базы данных, откуда мы могли бы извлекать объекты `Ingredient`, поэтому конструктор `IngredientByIdConverter` создает экземпляр `Map` с ключами типа `String`, которые служат идентификаторами ингредиентов, и значениями типа `Ingredient`, представляющими сами объекты. В главе 3 мы изменим этот конвертер так, чтобы он извлекал ингредиенты из базы данных. Метод `convert()` просто принимает строку, служащую идентификатором ингредиента, и использует ее для поиска соответствующего ингредиента в ассоциативном массиве `Map`.

Обратите внимание, что `IngredientByIdConverter` снабжен аннотацией `@Component`, то есть он обнаруживается механизмом сканирования и создается как bean-компонент в контексте приложения Spring. Механизм автоконфигурации Spring Boot обнаружит этот и все другие bean-компоненты, реализующие интерфейс `Converter`, и автоматически зарегистрирует их для использования в Spring MVC, когда потребуется преобразовать параметры запроса в свойства.

В настоящий момент метод `processTaco()` не делает ничего особенного с объектом `Taco`. Но пусть вас это не беспокоит. В главе 3 мы добавим дополнительную логику, которая сохранит полученный объект `Taco` в базу данных.

Так же как `showDesignForm()`, метод `processTaco()` возвращает строковое значение. И так же как в случае с `showDesignForm()`, возвращае-

мое значение определяет имя представления, которое будет отправлено пользователю. Единственное существенное отличие – значение, возвращаемое из `processTaco()`, имеет префикс "redirect:", сообщающий, что это представление с перенаправлением, то есть после завершения `processTaco()` браузер пользователя должен открыть другую страницу, отправив запрос GET с путем `/orders/curren`.

Идея состоит в том, чтобы после создания тако перенаправить пользователя на форму заказа, откуда он сможет сделать заказ на доставку своего тако. Но у нас пока нет контроллера, который будет обрабатывать запросы с путем `/orders/curren`.

Теперь, после знакомства с аннотациями `@Controller`, `@RequestMapping` и `@GetMapping`, вы с легкостью сможете сами определить такой контроллер. Он мог бы выглядеть примерно так, как показано в листинге 2.8.

Листинг 2.8 Контроллер, представляющий форму заказа тако

```
package tacos.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

import lombok.extern.slf4j.Slf4j;
import tacos.TacoOrder;

@Slf4j
@Controller
@RequestMapping("/orders")
@SessionAttributes("tacoOrder")
public class OrderController {

    @GetMapping("/current")
    public String orderForm() {
        return "orderForm";
    }
}
```

И снова мы используем аннотацию `@Slf4j` из библиотеки Lombok, чтобы автоматически создать объект `Logger` во время компиляции. Мы используем его чуть ниже, чтобы зарегистрировать в журнале детали отправленного заказа.

Аннотация `@RequestMapping` на уровне класса сообщает, что любые методы обработки запросов в этом контроллере будут обрабатывать запросы с путями, начинающимися с `/orders`. В сочетании с ней аннотация `@GetMapping` на уровне метода `orderForm()` указывает, что этот метод будет обрабатывать HTTP-запросы GET с путем `/orders/curren`.

Сам метод `orderForm()` чрезвычайно прост и просто возвращает логическое имя представления `orderForm`. Как только у нас появится возможность сохранять созданные тако в базе данных, мы вернемся к этому методу и изменим его так, чтобы он заполнял модель списком объектов Taco в порядке их создания.

Представление `orderForm` реализовано в виде шаблона Thymeleaf с именем `orderForm.html`, который показан в листинге 2.9.

Листинг 2.9 Форма заказа тако

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Taco Cloud</title>
    <link rel="stylesheet" th:href="@{/styles.css}" />
</head>

<body>

    <form method="POST" th:action="@{/orders}" th:object="${tacoOrder}">
        <h1>Order your taco creations!</h1>

        <h3>Your tacos in this order:</h3>
        <a th:href="@{/design}" id="another">Design another taco</a><br/>
        <ul>
            <li th:each="taco : ${tacoOrder.tacos}">
                <span th:text="${taco.name}">taco name</span></li>
        </ul>

        <h3>Deliver my taco masterpieces to...</h3>
        <label for="deliveryName">Name: </label>
        <input type="text" th:field="*{deliveryName}" />
        <br/>

        <label for="deliveryStreet">Street address: </label>
        <input type="text" th:field="*{deliveryStreet}" />
        <br/>

        <label for="deliveryCity">City: </label>
        <input type="text" th:field="*{deliveryCity}" />
        <br/>

        <label for="deliveryState">State: </label>
        <input type="text" th:field="*{deliveryState}" />
        <br/>

        <label for="deliveryZip">Zip code: </label>
        <input type="text" th:field="*{deliveryZip}" />
        <br/>
```

```
<h3>Here's how I'll pay...</h3>
<label for="ccNumber">Credit Card #: </label>
<input type="text" th:field="*{ccNumber}"/>
<br/>

<label for="ccExpiration">Expiration: </label>
<input type="text" th:field="*{ccExpiration}"/>
<br/>

<label for="ccCVV">CVV: </label>
<input type="text" th:field="*{ccCVV}"/>
<br/>

<input type="submit" value="Submit Order"/>
</form>
</body>
</html>
```

Большую часть представления `orderForm.html` составляет типичная разметка HTML/Thymeleaf, в которой почти нечего отметить. Она начинается с перечисления тако, добавленных в заказ с помощью `th:each`, и затем отображает форму заказа.

Но обратите внимание, что тег `<form>` здесь отличается от тега `<form>` в листинге 2.5: он определяет атрибут `action`. Без этого атрибута форма отправит HTTP-запрос POST обратно на тот же URL. Но здесь мы указали, что форма должна быть отправлена HTTP-запросом POST с путем `/orders` (это сделано с помощью оператора `@{...}` определения контекстно-зависимого пути).

Поэтому нам нужно будет добавить в класс `OrderController` еще один метод, обрабатывающий POST-запросы с путем `/orders`. Возможность сохранять заказы у нас появится только в следующей главе, поэтому сейчас мы просто сообщим пользователю, что заказ принят, как показано в листинге 2.10.

Листинг 2.10 Обработка отправки заказа

```
@PostMapping
public String processOrder(TacoOrder order,
    SessionStatus sessionStatus) {
    log.info("Order submitted: {}", order);
    sessionStatus.setComplete();

    return "redirect:/";
}
```

Когда приложение вызовет метод `processOrder()` для обработки отправленного заказа, ему будет передан объект `TacoOrder` со значениями свойств, полученными из полей отправленной формы. `TacoOrder`, как и `Taco`, – довольно простой класс, несущий информацию о заказе.

В настоящее время метод `processOrder()` просто регистрирует объект `TacoOrder` в журнале, но в следующей главе вы увидите, как

сохранить его в базе данных. Прежде чем завершиться, метод `processOrder()` вызывает метод `setComplete()` объекта `SessionStatus`, переданного в параметре. Первоначально объект `TacoOrder` был создан и помещен в сеанс, когда пользователь создал свой первый тако. Вызывая `setComplete()`, мы гарантируем, что сеанс будет очищен и готов к приему нового заказа, когда пользователь создаст тако.

Теперь, создав контроллер `OrderController` и представление формы заказа, мы готовы опробовать его. Откройте в браузере страницу `http://localhost:8080/design`, выберите ингредиенты для тако и щелкните на кнопке **Submit Your Taco** (Отправить рецепт тако). Вы должны увидеть форму, похожую на изображенную на рис. 2.4.

localhost

Order your taco creations!

Taco Cloud

[Design another taco](#)

Deliver my taco masterpieces to...

Name:

Street address:

City:

State:

Zip code:

Here's how I'll pay...

Credit Card #:

Expiration:

CVV:

Рис. 2.4 Форма заказа тако

Заполните поля формы и щелкните на кнопке **Submit Order** (Отправить заказ). Затем загляните в журнал приложения, где должна по-

явиться информация о вашем заказе. Когда я попробовал выполнить эти действия, то у меня запись в журнале выглядела примерно так (переформатирована, чтобы уместить ее по ширине страницы):

```
Order submitted: TacoOrder(deliveryName=Craig Walls, deliveryStreet=1234 7th Street, deliveryCity=Somewhere, deliveryState=Who knows?, deliveryZip=zipzap, ccNumber=Who can guess?, ccExpiration=Some day, ccCVV=See-vee-vee, tacos=[Taco(name=Awesome Sauce, ingredients=[Ingredient(id=FLTO, name=Flour Tortilla, type=WRAP), Ingredient(id=GRBF, name=Ground Beef, type=PROTEIN), Ingredient(id=CHED, name=Cheddar, type=CHEESE), Ingredient(id=TMT0, name=Diced Tomatoes, type=VEGGIES), Ingredient(id=SLSA, name=Salsa, type=SAUCE), Ingredient(id=SRCR, name=Sour Cream, type=SAUCE)]), Taco(name=Quesoriffic, ingredients=[Ingredient(id=FLTO, name=Flour Tortilla, type=WRAP), Ingredient(id=CHED, name=Cheddar, type=CHEESE), Ingredient(id=JACK, name=Monterrey Jack, type=CHEESE), Ingredient(id=TMT0, name=Diced Tomatoes, type=VEGGIES), Ingredient(id=SRCR, name=Sour Cream, type=SAUCE)]])])
```

Похоже, что метод `processOrder()` благополучно выполнил свою работу, обработав отправленную ему форму и записав сведения о заказе в журнал. Но если внимательно посмотреть на запись в моем журнале, то можно заметить, что она содержит немного ошибочной информации. Часть данных, которые я ввел в форму, нельзя назвать верными. Давайте добавим проверку данных, чтобы гарантировать, что клиент сможет прислать только верные данные.

2.3 Проверка данных в форме

При создании нового рецепта пользователь может не выбрать ингредиенты или не указать имя своего творения. При отправке заказа он может не заполнить обязательные поля адреса или ввести недействительный номер кредитной карты. Как быть в таких случаях?

В настоящее время ничто не помешает пользователю создать тако без ингредиентов, отправить заказ с пустым адресом доставки или вместо номера кредитной карты ввести слова своей любимой песни. А это значит, что мы должны реализовать проверку этих полей.

Один из способов организовать такую проверку – замусорить методы `processTaco()` и `processOrder()` множеством операторов `if/then`, проверяющих каждое поле на соответствие установленным правилам. Но тогда код получился бы слишком громоздким и трудночитаемым.

К счастью, Spring поддерживает API проверки JavaBean (также известный как JSR 303; <https://jcp.org/en/jsr/detail?id=303>). Он упрощает объявление правил проверки и не требует явного описания логики в коде приложения.

Чтобы организовать проверку в Spring MVC, необходимо:

- добавить в сборку начальную зависимость от Spring Validation;
- объявить правила проверки для проверяемого класса: в частности, для класса `Taco`;

- указать, в каких методах каких контроллеров должна выполнятьсь проверка: в данном случае в методе `processTaco()` контроллера `DesignTacoController` и в методе `processOrder()` контроллера `OrderController`;
- изменить представления форм для отображения ошибок, выявленных при проверке.

Интерфейс проверки Validation API предлагает несколько аннотаций, которыми можно снабдить свойства объектов данных и тем самым объявить правила проверки. Реализация Validation API от Hibernate включает еще больше аннотаций проверки. В проект можно добавить обе реализации, включив в спецификацию сборки начальную зависимость Spring Validation. Для этого достаточно установить флагок **Validation** (Проверка) в разделе **I/O** (Ввод/вывод) в мастере Spring Boot Starter, но если вы предпочитаете править настройки сборки вручную, то добавьте следующие строки в файл `pom.xml`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

Или если вы используете Gradle, то добавьте зависимость так:

```
implementation 'org.springframework.boot:spring-boot-starter-validation'
```

Действительно ли необходимо подключать начальную зависимость Validation API?

В ранних версиях Spring Boot начальная зависимость Spring Validation подключалась автоматически вместе с веб-фреймворком. Но начиная с версии Spring Boot 2.3.0 интерфейс Validation API нужно добавлять в сборку явно.

Теперь, добавив начальную зависимость, давайте посмотрим, как с помощью аннотаций выполнить проверку полученных объектов Taco и TacoOrder.

2.3.1 Объявление правил проверки

Получив экземпляр класса Taco, мы должны убедиться, что клиент не забыл заполнить поле с названием своего рецепта и выбрал хотя бы один ингредиент. В листинге 2.11 показан дополненный класс Taco, в котором для выполнения упомянутых проверок использованы аннотации `@NotNull` и `@Size`.

Листинг 2.11 Добавление проверок в класс данных Taco

```
package tacos;
import java.util.List;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;
import lombok.Data;

@Data
public class Taco {

    @NotNull
    @Size(min=5, message="Name must be at least 5 characters long")
    private String name;

    @NotNull
    @Size(min=1, message="You must choose at least 1 ingredient")
    private List<Ingredient> ingredients;
}
```

Обратите внимание, что согласно требованиям свойство `name` не только должно быть непустым, но иметь длину не менее пяти символов.

Для проверки формы заказа мы должны применить аннотации к классу `TacoOrder`. В отношении свойств с адресом мы должны убедиться, что клиент заполнил их все, для чего используем аннотацию `@NotNull`.

А вот проверка полей, связанных с оплатой, организуется немного экзотичнее. Мы должны убедиться, что свойство `ccNumber` не только не пустое, но и содержит значение, которое похоже на действительный номер кредитной карты. Свойство `ccExpiration` должно быть заполнено в соответствии с форматом ММ/ГГ (две цифры месяца и две цифры года), а свойство `ccCVV` должно содержать трехзначное число. Чтобы реализовать такие проверки, нужно использовать несколько других аннотаций из Validation API и позаимствовать аннотацию проверки из коллекции аннотаций Hibernate Validator. В листинге 2.12 показано, какие изменения необходимо внести в класс `TacoOrder` для реализации этих проверок.

Листинг 2.12 Проверка полей заказа

```
package tacos;
import javax.validation.constraints.Digits;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.CreditCardNumber;
import java.util.List;
import java.util.ArrayList;
import lombok.Data;
```

```

@Data
public class TacoOrder {

    @NotBlank(message="Delivery name is required")
    private String deliveryName;

    @NotBlank(message="Street is required")
    private String deliveryStreet;

    @NotBlank(message="City is required")
    private String deliveryCity;

    @NotBlank(message="State is required")
    private String deliveryState;

    @NotBlank(message="Zip code is required")
    private String deliveryZip;

    @CreditCardNumber(message="Not a valid credit card number")
    private String ccNumber;

    @Pattern(regexp="^(0[1-9]|1[0-2])([\\ \\ /])([2-9][0-9])$",
              message="Must be formatted MM/YY")
    private String ccExpiration;

    @Digits(integer=3, fraction=0, message="Invalid CVV")
    private String ccCVV;

    private List<Taco> tacos = new ArrayList<>();

    public void addTaco(Taco taco) {
        this.tacos.add(taco);
    }
}

```

Как видите, свойство `ccNumber` снабжено аннотацией `@CreditCardNumber`. Эта аннотация требует, чтобы значение свойства было действительным номером кредитной карты, прошедшим проверку алгоритмом Луна (<https://creditcardvalidator.org/articles/luhn-algorithm>). Это предотвращает ошибки пользователя и преднамеренный ввод неверных данных, но не гарантирует, что номер кредитной карты соответствует действующему счету и этот счет позволяет списывать средства.

К сожалению, не существует готовой аннотации для проверки формата ММ/ГГ свойства `ccExpiration`. Поэтому я использовал аннотацию `@Pattern`, добавив в нее регулярное выражение, проверяющее соответствие значения желаемому формату. Если вам интересно, как расшифровать это регулярное выражение, я рекомендую ознакомиться с любым из множества онлайн-руководств по регулярным выражениям, например <http://www.regular-expressions.info/>. Синтаксис регулярных выражений – это тайная магия, и его обсуждение выходит за рамки нашей книги. Наконец, свойство `ccCVV` снабжено

аннотацией `@Digits`, гарантирующей, что значение содержит ровно три цифры.

Все аннотации механизма проверки включают атрибут `message`, в котором можно определить сообщение для отображения пользователю, если введенная им информация не соответствует требованиям правил проверки.

2.3.2 Выполнение проверки после привязки

Теперь, когда мы организовали проверку экземпляров Taco и TacoOrder, вернемся к каждому из контроллеров и укажем, что проверка должна выполняться, когда формы передаются для обработки соответствующим методам-обработчикам.

Чтобы проверить отправленный экземпляр Taco, нужно добавить аннотацию `@Valid` из Validation API перед аргументом Taco в объявлении метода `processTaco()` контроллера `DesignTacoController`, как показано в листинге 2.13.

Листинг 2.13 Проверка отправленного экземпляра Taco

```
import javax.validation.Valid;
import org.springframework.validation.Errors;
...
@PostMapping
public String processTaco(
    @Valid Taco taco, Errors errors,
    @ModelAttribute TacoOrder tacoOrder) {
    if (errors.hasErrors()) {
        return "design";
    }

    tacoOrder.addTaco(taco);
    log.info("Processing taco: {}", taco);

    return "redirect:/orders/current";
}
```

Аннотация `@Valid` требует выполнить проверку отправленного объекта Taco после его привязки к данным в отправленной форме, но до начала выполнения тела метода `processTaco()`. Если обнаружатся какие-либо ошибки, то сведения о них будут зафиксированы в объекте `Errors`, который передается в `processTaco()`. Первые несколько строк в `processTaco()` проверяют наличие ошибок, вызывая метод `hasErrors()` объекта `Errors`. Если ошибки есть, то метод `processTaco()` завершает работу без обработки Taco и возвращает имя представления "design", чтобы повторно отобразить форму.

Для проверки отправленного объекта `TacoOrder` необходимо внести аналогичные изменения в метод `processOrder()` контроллера `OrderController`, как показано в листинге 2.14.

Листинг 2.14 Проверка отправленного экземпляра TacoOrder

```

@PostMapping
public String processOrder(@Valid TacoOrder order, Errors errors,
    SessionStatus sessionStatus) {
    if (errors.hasErrors()) {
        return "orderForm";
    }

    log.info("Order submitted: {}", order);
    sessionStatus.setComplete();

    return "redirect:/";
}

```

В обоих случаях, если ошибок не обнаружится, методы продолжат обработку отправленных данных. При наличии ошибок запрос будет передан в представление формы, чтобы дать пользователю возможность исправить эти ошибки.

Но как пользователь узнает, какие ошибки требуют исправления? Если явно не указать, какие поля заполнены с ошибками, пользователю останется только гадать, как правильно заполнить форму.

2.3.3 Отображение сообщений об ошибках

Thymeleaf предлагает удобный доступ к объекту `Errors` через свойство `fields` и его атрибут `th:errors`. Например, чтобы отобразить сообщение об ошибке в поле номера кредитной карты, можно добавить в шаблон формы элемент ``, использующий ссылки на ошибки, как показано в листинге 2.15.

Листинг 2.15 Отображение сообщений об ошибках

```

<label for="ccNumber">Credit Card #: </label>
<input type="text" th:field="*{ccNumber}" />
<span class="validationError"
    th:if="${#fields.hasErrors('ccNumber')}"
    th:errors="*{ccNumber}">CC Num Error</span>

```

Помимо атрибута `class`, который можно использовать для оформления сообщения об ошибке, чтобы привлечь внимание пользователя, элемент `` использует также атрибут `th:if`, чтобы решить, следует ли сделать себя видимым. Метод `hasErrors()` свойства `fields` проверяет наличие ошибок в поле `ccNumber`. Если ошибки есть, то `` будет видим.

Атрибут `th:errors` ссылается на поле `ccNumber`, и если в этом поле обнаружились ошибки, то он заменит содержимое элемента `` сообщением, сгенерированным во время проверки.

Если добавить подобные теги `` во все поля формы заказа, то, допустив ошибки, пользователь получил бы обратно форму, похожую на изображенную на рис. 2.5. В этом случае сообщения об ошибках указывают, что поля с названиями рецепта и города, а также поле почтового индекса не были заполнены, а поля, связанные с оплатой, не соответствуют критериям проверки.

The screenshot shows a web browser window with the URL `localhost`. The page title is "Order your taco creations!". In the center is a large illustration of a taco filled with meat, cheese, and vegetables, with the words "Taco Cloud" written across it. Below the illustration, there is a message: "Please correct the problems below and resubmit." To the right of this message is a link: "Design another taco".
Deliver my taco masterpieces to...
Name: Name is required
Street address: 1234 7th Street
City: City is required
State: VT
Zip code: Zip code is required
Here's how I'll pay...
Credit Card #: Who can guess? Not a valid credit card number
Expiration: Some day Must be formatted MM/YY
CVV: see-vee-vee Invalid CVV

Рис. 2.5 Ошибки проверки, отображаемые в форме заказа

Теперь ваши контроллеры в Taco Cloud не только отображают и фиксируют введенные данные, но также проверяют их на соответствие некоторым простым правилам. А сейчас вернемся немного назад и рассмотрим альтернативную реализацию `HomeController`.

2.4 Работа с контроллерами представлений

На данный момент мы написали три контроллера для приложения Taco Cloud. Каждый контроллер служит определенной цели, но все они следуют одной модели программирования:

- все контроллеры снабжены аннотацией `@Controller`, чтобы механизм сканирования в Spring мог обнаружить их и создать bean-компоненты в контексте приложения;
- все контроллеры, кроме `HomeController`, снабжены аннотацией `@RequestMapping` на уровне класса, определяющей базовый шаблон запросов, которые будет обрабатывать контроллер;
- все контроллеры имеют один или несколько методов, снабженных аннотацией `@GetMapping` или `@PostMapping`, определяющей тип обрабатываемых запросов.

Большинство ваших контроллеров будут следовать этому шаблону. Но если контроллер достаточно прост, как `HomeController`, то, чтобы не заполнять входную модель данными, можно определить контроллер другим способом. Взгляните на листинг 2.16, где показано, как объявить контроллер представления – контроллер, который ничего не делает, кроме как передает запрос в представление.

Листинг 2.16 Объявление контроллера представления

```
package tacos.web;

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 WebConfig implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
    }
}
```

Первое, что следует отметить в классе `WebConfig`, – он реализует интерфейс `WebMvcConfigurer`. Этот интерфейс определяет несколько методов настройки Spring MVC. Несмотря на то что это интерфейс, он предоставляет реализации по умолчанию для всех методов, поэтому нам остается переопределить только те методы, которые нам нужны. В этом случае мы переопределили `addViewControllers()`.

Метод `addViewControllers()` получает экземпляр `ViewControllerRegistry`, с помощью которого можно зарегистрировать один или несколько контроллеров представлений. Здесь мы вызываем `addViewController()` с аргументом `"/"`, определяющим путь в запросах GET, которые должен обрабатывать этот контроллер представления. Этот метод возвращает объект `ViewControllerRegistration`, для которого мы тут же вызываем `setViewName()`, чтобы указать имя `home` представления, которому должны передаваться запросы `"/"`.

Контроллер `HomeController` можно также заменить несколькими строками в классе конфигурации, после чего удалить `HomeController`, и приложение будет действовать так же, как раньше. Единственное, что при этом потребуется изменить, – вернуться к классу `HomeControllerTest`, представленному в главе 1, и удалить ссылку на `HomeController` из аннотации `@WebMvcTest`, чтобы тестовый класс компилировался без ошибок.

Здесь мы создали новый класс конфигурации `WebConfig` для объявления контроллера представления. Но, вообще говоря, любой класс конфигурации может реализовать интерфейс `WebMvcConfigurer` и переопределить метод `addViewController`. Например, мы могли бы добавить такое же объявление контроллера представления в загрузочный класс `TacoCloudApplication`:

```
@SpringBootApplication
public class TacoCloudApplication implements WebMvcConfigurer {

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

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/").setViewName("home");
    }
}
```

Наследуя существующий класс конфигурации, можно избежать необходимости создания нового класса конфигурации и уменьшить количество артефактов в проекте. Но я предпочитаю создавать новый класс для каждого типа конфигурации (веб, данные, безопасность и т. д.), сохраняя при этом конфигурацию начальной загрузки приложения простой и ясной.

До сих пор для реализации представлений, которым контроллеры пересыпают запросы, мы использовали механизм шаблонов Thymeleaf. Мне очень нравится Thymeleaf, но, возможно, вы предпочтете другой механизм шаблонов. Давайте посмотрим, какие иные средства создания представлений поддерживает Spring.

2.5 Выбор механизма шаблонов для создания представлений

В большинстве случаев выбор библиотеки шаблонов представлений зависит от личного вкуса. Spring – достаточно гибкий фреймворк и поддерживает множество распространенных механизмов шаблонов. За некоторыми редкими исключениями, выбранная вами библиотека шаблонов сама не будет знать, что она работает со Spring¹.

В табл. 2.2 перечислены некоторые механизмы шаблонов, поддерживаемые механизмом автоконфигурации Spring Boot.

Таблица 2.2 Поддерживаемые механизмы шаблонов

Механизм шаблонов	Начальная зависимость Spring Boot
FreeMarker	spring-boot-starter-freemarker
Шаблоны Groovy	spring-boot-starter-groovy-templates
JavaServer Pages (JSP)	Нет (поддерживается Tomcat или Jetty)
Mustache	spring-boot-starter-mustache
Thymeleaf	spring-boot-starter-thymeleaf

В общем случае вы выбираете понравившуюся библиотеку шаблонов представлений, добавляете ее как зависимость в спецификацию сборки и начинаете писать шаблоны в каталоге */templates* (в каталоге *src/main/resources* в проекте Maven или Gradle). Spring Boot автоматически обнаружит выбранную вами библиотеку шаблонов и настроит компоненты, необходимые для обслуживания представлений, на которые ссылаются ваши контроллеры Spring MVC.

Именно так мы и поступили, подключив библиотеку Thymeleaf к приложению Taco Cloud. В главе 1 мы установили флагок **Thymeleaf** в форме инициализации проекта, в результате чего в спецификацию конфигурации в *root.xml* была добавлена зависимость Spring Boot Thymeleaf. В момент запуска приложения механизм автоконфигурации Spring Boot обнаруживает Thymeleaf и автоматически настраивает bean-компоненты Thymeleaf в контексте приложения. Нам остается только начать писать шаблоны в */templates*.

Если вы предпочитаете использовать другую библиотеку шаблонов, то просто выберите ее при инициализации проекта или отредактируйте спецификацию сборки существующего проекта, чтобы задействовать другую библиотеку шаблонов.

Например, предположим, что мы решили использовать Mustache вместо Thymeleaf. Никаких проблем. Просто откроем файл проекта *root.xml* и заменим строки

```
<dependency>
  <groupId>org.springframework.boot</groupId>
```

¹ Одно из таких исключений – диалект Thymeleaf для Spring Security, о котором мы поговорим в главе 5.

```
<artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

на

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-mustache</artifactId>
</dependency>
```

После этого, конечно же, мы должны переписать все шаблоны на языке Mustache. Специфика работы с Mustache (или любым другим языком шаблонов) выходит за рамки нашей книги, но, чтобы дать вам представление о том, чего можно ожидать, ниже приводится фрагмент шаблона Mustache, который отображает один из ингредиентов в форме создания рецепта тако:

```
<h3>Designate your wrap:</h3>
{{#wrap}}
<div>
  <input name="ingredients" type="checkbox" value="{{id}}'' />
  <span>{{name}}</span><br/>
</div>
{{/wrap}}
```

Это аналог фрагмента Thymeleaf из раздела 2.1.3 на языке Mustache. Блок {{#wrap}} (который завершается {{/wrap}}) выполняет итерации по коллекции в атрибуте wrap запроса и преобразует каждый элемент в соответствующую разметку HTML. Теги {{id}} и {{name}} – это ссылки на свойства id и name элемента (который должен быть экземпляром Ingredient).

Возможно, вы заметили в табл. 2.2, что JSP не требует наличия каких-либо зависимостей в спецификации сборки. Это связано с тем, что сам контейнер сервлетов (по умолчанию Tomcat) реализует спецификацию JSP, соответственно, ему не нужны дополнительные зависимости.

Но тут есть один подвох для тех, кто решит использовать JSP. Как оказывается, контейнеры сервлетов Java, включая встроенные контейнеры Tomcat и Jetty, обычно ищут JSP где-то в каталоге /WEB-INF. Но если вы создаете приложение в виде выполняемого файла JAR, то удовлетворить это требование невозможно. Поэтому JSP подходит только для случаев, когда приложение создается в виде файла WAR и развертывается в традиционном контейнере сервлетов. Если вы создаете выполняемый файл JAR, то должны выбирать Thymeleaf, FreeMarker или один из других вариантов, перечисленных в табл. 2.2.

2.5.1 Кеширование шаблонов

По умолчанию шаблоны анализируются только один раз – при первом использовании, – и результаты этого анализа кешируются для

последующего использования. Это отличное решение для промышленного использования, потому что предотвращает избыточный анализ шаблонов при обработке каждого запроса и тем самым повышает производительность.

Однако эта особенность не так хороша во время разработки. Допустим, вы запускаете свое приложение, попадаете на страницу составления рецепта тако и решаете внести в нее несколько изменений. Когда вы обновите страницу в веб-браузере, то увидите прежнюю версию. Единственный способ добиться, чтобы изменения вступили в силу, – перезапустить приложение, что довольно неудобно.

К счастью, мы можем отключить кеширование. Для этого нужно лишь присвоить `false` свойству шаблона, отвечающему за кеширование. В табл. 2.3 перечислены свойства управления кешированием для каждой из поддерживаемых библиотек шаблонов.

Таблица 2.3 Свойства управления кешированием шаблонов

Механизм шаблонов	Имя свойства, управляющего кешированием
FreeMarker	<code>spring.freemarker.cache</code>
Шаблоны Groovy	<code>spring.groovy.template.cache</code>
Mustache	<code>spring.mustache.cache</code>
Thymeleaf	<code>spring.thymeleaf.cache</code>

По умолчанию всем этим свойствам присваивается значение `true`, чтобы включить кеширование. Вы можете отключить кеширование для выбранного механизма шаблонов, присвоив этому свойству `false`. Например, чтобы отключить кеширование шаблонов Thymeleaf, добавьте следующую строку в `application.properties`:

```
spring.thymeleaf.cache=false
```

Единственная загвоздка в том, что вы должны обязательно удалить эту строку (или установить в ней значение `true`) перед развертыванием приложения в промышленном окружении. Как вариант это свойство можно установить в профиле. (О профилях мы поговорим в главе 6.)

Гораздо проще использовать Spring Boot DevTools, как мы решили сделать в главе 1. Кроме всего прочего, DevTools отключит кеширование любых библиотек шаблонов на этапе разработки, но отключает себя (и тем самым обеспечивает автоматическое включение кеширования шаблонов) при развертывании приложения.

Итоги

- Spring предлагает мощный веб-фреймворк под названием Spring MVC, который можно использовать для разработки веб-интерфейса приложения Spring.

- Spring MVC основан на аннотациях, что позволяет объявлять методы обработки запросов с помощью таких аннотаций, как `@RequestMapping`, `@GetMapping` и `@PostMapping`.
- Большинство методов обработки запросов возвращают логическое имя представления, такого как шаблон Thymeleaf, которому передается запрос (вместе с любыми данными модели).
- Spring MVC поддерживает проверку через JavaBean Validation API и такие реализации Validation API, как Hibernate Validator.
- Контроллеры представлений можно зарегистрировать с помощью `addViewController` в классе `WebMvcConfigurer`, чтобы обрабатывать HTTP-запросы GET, для которых не требуются данные или какая-то дополнительная обработка.
- Кроме Thymeleaf, Spring поддерживает множество других механизмов шаблонов, включая FreeMarker, шаблоны Groovy и Mustache.

Работа с данными

В этой главе рассматриваются следующие темы:

- использование Spring `JdbcTemplate`;
- создание репозиториев Spring Data JDBC;
- объявление репозиториев JPA с помощью Spring Data.

Большинство приложений предлагают не только красивый пользовательский интерфейс. Конечно, пользовательский интерфейс играет немаловажную роль во взаимодействии человека с приложением, но именно данные, которые он представляет и хранит, отделяют приложения от статических веб-сайтов.

В приложении Taco Cloud мы должны иметь возможность хранить информацию об ингредиентах, рецептах тако и заказах. Без базы данных для хранения этой информации мы не сможем продвинуть приложение дальше той точки, что достигли в главе 2.

В этой главе мы добавим в приложение Taco Cloud поддержку хранения данных. Для начала используем поддержку JDBC (Java Database Connectivity – соединение с базами данных) в Spring, чтобы исключить шаблонный код. Затем для работы с репозиториями данных задействуем JPA (Java Persistence API), чтобы упростить код еще больше.

3.1 Чтение и запись данных с помощью JDBC

В течение десятилетий реляционные базы данных и SQL занимали доминирующее положение в сфере хранения данных. Несмотря на то что в последние годы появилось много альтернативных типов баз данных, реляционные базы данных по-прежнему считаются лучшим выбором на роль хранилища данных общего назначения и вряд ли уступят свои позиции в ближайшее время.

В отношении работы с реляционными данными у Java-разработчиков есть несколько вариантов на выбор. Двумя наиболее распространенными являются JDBC и JPA. Spring поддерживает оба варианта, предлагая свои абстракции, упрощающие работу с JDBC и JPA. В этом разделе мы сосредоточимся на поддержке JDBC, а затем, в разделе 3.2, рассмотрим поддержку JPA.

Поддержка JDBC в Spring уходит корнями в класс `JdbcTemplate`. Этот класс позволяет разработчикам посыпать SQL-запросы реляционной базе данных без лишних церемоний и шаблонов, характерных для работы с JDBC.

Чтобы получить представление о работе `JdbcTemplate`, рассмотрим пример выполнения простого запроса в Java без `JdbcTemplate` (листинг 3.1).

Листинг 3.1 Выполнение запроса к базе данных без `JdbcTemplate`

```
@Override
public Optional<Ingredient> findById(String id) {
    Connection connection = null;
    PreparedStatement statement = null;
    ResultSet resultSet = null;
    try {
        connection = dataSource.getConnection();
        statement = connection.prepareStatement(
            "select id, name, type from Ingredient where id=?");
        statement.setString(1, id);
        resultSet = statement.executeQuery();
        Ingredient ingredient = null;
        if(resultSet.next()) {
            ingredient = new Ingredient(
                resultSet.getString("id"),
                resultSet.getString("name"),
                Ingredient.Type.valueOf(resultSet.getString("type")));
        }
        return Optional.of(ingredient);
    } catch (SQLException e) {
        // ??? Что делать, если возникнет исключение ???
    } finally {
        if (resultSet != null) {
            try {
                resultSet.close();
            }
```

```

        } catch (SQLException e) {}
    }
    if (statement != null) {
        try {
            statement.close();
        } catch (SQLException e) {}
    }
    if (connection != null) {
        try {
            connection.close();
        } catch (SQLException e) {}
    }
}
return Optional.empty();
}

```

Где-то в недрах листинга 3.1 есть пара строк, которые запрашивают ингредиенты из базы данных. Но я готов поспорить, что вам будет не-легко отыскать иголку запроса в стоге сена JDBC. Он окружен кодом, который создает соединение и запрос, а затем закрывает соединение, запрос и результирующий набор.

Что особенно неприятно, многое в этой процедуре, включающей открытие соединения, создание и выполнение запроса и освобождение ресурсов после запроса, может пойти не так. И это требует перехватывать исключение `SQLException`, что может помочь, а может и не помочь, выяснить причину и решить проблему.

`SQLException` – это контролируемое исключение, которое требует обработки в блоке `catch`. Но типичные проблемы, такие как сбой при открытии соединения с базой данных или во время выполнения запроса, невозможно решить в блоке `catch`, и поэтому исключение часто генерируется вновь, чтобы передать его обработку функциям, находящимся выше в потоке выполнения. А теперь взгляните на следующий метод (листинг 3.2), использующий `JdbcTemplate` из фреймворка Spring.

Листинг 3.2 Выполнение запроса к базе данных с `JdbcTemplate`

```

Spring.private JdbcTemplate jdbcTemplate;

public Optional<Ingredient> findById(String id) {
    List<Ingredient> results = jdbcTemplate.query(
        "select id, name, type from Ingredient where id=?",
        this::mapRowToIngredient,
        id);
    return results.size() == 0 ?
        Optional.empty() :
        Optional.of(results.get(0));
}

private Ingredient mapRowToIngredient(ResultSet row, int rowNum)
    throws SQLException {

```

```
        return new Ingredient(
            row.getString("id"),
            row.getString("name"),
            Ingredient.Type.valueOf(row.getString("type")));
    }
```

Код в листинге 3.2 явно намного проще, чем первоначальный вариант с использованием JDBC в листинге 3.1; здесь не создаются ни запросы, ни соединения. И после завершения не выполняется никаких заключительных операций. Наконец, полностью отсутствует обработка исключений, которые на этом уровне невозможно правильно обработать. Остался код, ориентированный исключительно на выполнение запроса (вызов метода `JdbcTemplate.query()`) и отображение результатов в объект `Ingredient` (производится методом `mapRowToIngredient()`).

Код в листинге 3.2 содержит только то, что необходимо, чтобы с помощью `JdbcTemplate` сохранить и прочитать данные в приложении Taco Cloud. Давайте продолжим реализацию поддержки хранения данных в приложении с использованием механизма хранения JDBC. Для начала внесем несколько изменений в объекты данных.

3.1.1 Подготовка объектов данных к хранению

В объекты, предназначенные для хранения в базе данных, обычно рекомендуется добавлять поле, однозначно идентифицирующее каждый объект. В нашем классе `Ingredient` уже есть поле `id`, но нам необходимо также добавить поля `id` в классы `Taco` и `TacoOrder`.

Кроме того, иногда может оказаться полезным знать, когда создан объект `Taco` и когда был сделан заказ `TacoOrder`. То есть мы добавим также в каждый класс поле, хранящее дату и время сохранения конкретного объекта. В листинге 3.3 показаны новые поля `id` и `createdAt` в классе `Taco`.

Листинг 3.3 Добавление полей идентификатора и времени создания в класс `Taco`

```
@Data
public class Taco {
    private Long id;
    private Date createdAt = new Date();
    // ...
}
```

Поскольку мы используем библиотеку Lombok, которая автоматически создает методы доступа во время компиляции, то больше ничего добавлять не нужно, достаточно просто объявить свойства `id`

и `createdAt`. Они автоматически получат все необходимые методы. Подобные изменения внесем и в класс `TacoOrder`, как показано ниже:

```
@Data
public class TacoOrder implements Serializable {

    private static final long serialVersionUID = 1L;

    private Long id;

    private Date placedAt;

    // ...
}
```

И снова Lombok автоматически генерирует методы доступа, поэтому от нас требуется только добавить в `TacoOrder` нужные свойства. Если по какой-то причине вы решили отказаться от использования Lombok, то вам придется самим написать все необходимые методы.

Теперь наши классы данных готовы для постоянного хранения. Посмотрим, как объекты этих классов можно сохранять и извлекать из базы данных с помощью `JdbcTemplate`.

3.1.2 Использование `JdbcTemplate`

Прежде чем начать использовать `JdbcTemplate`, мы должны добавить соответствующую зависимость. Самый простой способ – добавить начальную зависимость Spring Boot JDBC в спецификацию сборки:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

Нам также понадобится база данных, где будут храниться данные. Для этапа разработки вполне подойдет встроенная база данных. Лично я предпочитаю встроенную базу данных H2, поэтому добавил в спецификацию сборки следующую зависимость:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

По умолчанию для базы данных генерируется случайное имя. Но это затрудняет определение и ввод вручную URL базы данных, если по какой-то причине вам понадобится подключиться к ней с помощью консоли H2 (которая при использовании Spring Boot DevTools доступна по адресу `http://localhost:8080/h2-console`). Поэтому определим

имя базы данных, установив пару свойств в файле *application.properties*, как показано ниже:

```
spring.datasource.generate-unique-name=false  
spring.datasource.name=tacocloud
```

Или, если хотите, переименуйте *application.properties* в *application.yml* и добавьте свойства в формате YAML:

```
spring:  
  datasource:  
    generate-unique-name: false  
    name: tacocloud
```

Выбор между форматами файла свойств и YAML – это вопрос личных предпочтений. Spring Boot с готовностью примет любой из них. Однако, учитывая структурированность и лучшую удобочитаемость формата YAML, далее в книге мы будем описывать свойства в формате YAML.

Присвоив свойству `spring.datasource.generate-unique-name` значение `false`, мы сообщаем фреймворку Spring, что он не должен генерировать уникальное случайное имя базы данных. Вместо этого следует использовать значение свойства `spring.datasource.name`. В нашем случае база данных будет называться "tacocloud". Соответственно, URL базы данных будет иметь вид: "jdbc:h2:mem:tacocloud". Вы сможете использовать его для подключения к базе данных в консоли H2.

Далее в книге я покажу, как настроить приложение для использования внешней базы данных, а пока перейдем к созданию репозитория, который будет извлекать и сохранять объекты `Ingredient`.

ОПРЕДЕЛЕНИЕ РЕПОЗИТОРИЕВ JDBC

Наш репозиторий для хранения объектов `Ingredient` должен поддерживать следующие операции:

- получение всех ингредиентов в виде коллекции объектов `Ingredient`;
- получение одного ингредиента по идентификатору;
- сохранение объекта `Ingredient`.

Следующий интерфейс `IngredientRepository` определяет эти три операции:

```
package tacos.data;  
  
import java.util.Optional;  
import tacos.Ingredient;  
  
public interface IngredientRepository {  
  
    Iterable<Ingredient> findAll();  
}
```

```

Optional<Ingredient> findById(String id);

Ingredient save(Ingredient ingredient);

}

```

Интерфейс наглядно демонстрирует, для чего вам нужен репозиторий ингредиентов, но нам все равно нужно написать реализацию `IngredientRepository`, которая использует `JdbcTemplate` для запросов к базе данных. Код в листинге 3.4 – это первый шаг в написании требуемой реализации.

Листинг 3.4 Начальная версия репозитория ингредиентов на основе `JdbcTemplate`

```

package tacos.data;
import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import tacos.Ingredient;

@Repository
public class JdbcIngredientRepository implements IngredientRepository {

    private JdbcTemplate jdbcTemplate;

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

    // ...
}

```

Как видите, класс `JdbcIngredientRepository` снабжен аннотацией `@Repository`. Это одна из нескольких стереотипных аннотаций, которые определяют фреймворк Spring, включая `@Controller` и `@Component`. Добавляя аннотацию `@Repository`, мы заявляем, что `JdbcIngredientRepository` должен автоматически обнаруживаться при сканировании компонентов и создаваться как bean-компонент в контексте приложения Spring.

Когда фреймворк Spring будет создавать bean-компонент `JdbcIngredientRepository`, он внедрит в него экземпляр `JdbcTemplate`. Это объясняется просто: когда имеется только один конструктор, Spring неявно применяет автоматическое связывание зависимостей через параметры этого конструктора. Если имеется более одного конструктора или если нужно, чтобы автоматическое связывание определялось явно, можно к конструктору добавить аннотацию `@Autowired`, как показано ниже:

```
@Autowired  
public JdbcIngredientRepository(JdbcTemplate jdbcTemplate) {  
    this.jdbcTemplate = jdbcTemplate;  
}
```

Конструктор сохраняет экземпляр `JdbcTemplate` в переменной экземпляра, которая будет использоваться в иных методах, реализующих чтение и запись объектов в базу данных. Коль скоро речь зашла о других методах, давайте взглянем на реализации `findAll()` и `findById()` (листинг 3.5).

Листинг 3.5 Реализация запросов к базе данных с помощью `JdbcTemplate`

```
@Override  
public Iterable<Ingredient> findAll() {  
    return jdbcTemplate.query(  
        "select id, name, type from Ingredient",  
        this::mapRowToIngredient);  
}  
  
@Override  
public Optional<Ingredient> findById(String id) {  
    List<Ingredient> results = jdbcTemplate.query(  
        "select id, name, type from Ingredient where id=?",
        this::mapRowToIngredient,
        id);  
    return results.size() == 0 ?
        Optional.empty() :
        Optional.of(results.get(0));  
}  
  
private Ingredient mapRowToIngredient(ResultSet row, int rowNum)
    throws SQLException {
    return new Ingredient(
        row.getString("id"),
        row.getString("name"),
        Ingredient.Type.valueOf(row.getString("type")));
}
```

Оба метода, `findAll()` и `findById()`, используют экземпляр `JdbcTemplate` аналогичным образом. Метод `findAll()`, как ожидается, возвращает коллекцию объектов и использует метод `query()` экземпляра `JdbcTemplate`. Метод `query()` принимает SQL-запрос, а также реализацию `RowMapper` из фреймворка Spring для отображения каждой записи из набора результатов в объект. Также метод `query()` принимает дополнительные аргументы со значениями для параметров в запросе. Но в данном случае таких параметров нет.

Запрос, который передается методу `findById()`, напротив, включает предложение `where` для сравнения значения столбца `id` со значением параметра `id`, передаваемого методу. Поэтому в вызов `query()`

передается один дополнительный аргумент `id`. При выполнении запроса символ `?` в инструкции SQL будет замещен значением этого аргумента.

Как показано в листинге 3.5, параметр `RowMapper` в обоих методах, `findAll()` и `findById()`, задается как ссылка на метод `mapRowToIngredient()`. Ссылки на методы и лямбда-выражения в Java удобно использовать при работе с `JdbcTemplate` вместо явной реализации `RowMapper`. Если по какой-то причине вам понадобится явно создать `RowMapper`, то используйте следующую реализацию `findById()`:

```
@Override
public Ingredient findById(String id) {
    return jdbcTemplate.queryForObject(
        "select id, name, type from Ingredient where id=?",
        new RowMapper<Ingredient>() {
            public Ingredient mapRow(ResultSet rs, int rowNum)
                throws SQLException {
                return new Ingredient(
                    rs.getString("id"),
                    rs.getString("name"),
                    Ingredient.Type.valueOf(rs.getString("type")));
            }
        }, id);
}
```

Чтение из базы данных – это только часть истории. В какой-то момент нам понадобится записать объекты в базу данных, чтобы потом их можно было получить обратно. Давайте посмотрим на реализацию метода `save()`.

ВСТАВКА НОВОЙ ЗАПИСИ

Метод `JdbcTemplate.update()` можно использовать для выполнения любых запросов, создающих новые или изменяющих существующие записи в базе данных. И как показано в листинге 3.6, его можно использовать для добавления новых записей в базу данных.

Листинг 3.6 Добавление новой записи с помощью `JdbcTemplate`

```
@Override
public Ingredient save(Ingredient ingredient) {
    jdbcTemplate.update(
        "insert into Ingredient (id, name, type) values (?, ?, ?)",
        ingredient.getId(),
        ingredient.getName(),
        ingredient.getType().toString());
    return ingredient;
}
```

Благодаря отсутствию необходимости отображать данные из `ResultSet` в объекты метод `update()` получился намного проще метода

query(). Он принимает только строку с SQL-запросом, а также значения для параметров в запросе. В данном случае запрос имеет три параметра, соответствующих последним трем параметрам метода save(), в которых передаются идентификатор, имя и тип ингредиента.

Теперь, завершив работу над JdbcIngredientRepository, мы можем внедрить его в DesignTacoController и использовать для предоставления списка объектов Ingredient вместо жестко «зашитых» значений (как было сделано в главе 2). В листинге 3.7 показаны изменения в DesignTacoController.

Листинг 3.7 Внедрение репозитория в контроллер и его использование

```
@Controller
@RequestMapping("/design")
@SessionAttributes("tacoOrder")
public class DesignTacoController {

    private final IngredientRepository ingredientRepo;

    @Autowired
    public DesignTacoController(
        IngredientRepository ingredientRepo) {
        this.ingredientRepo = ingredientRepo;
    }

    @ModelAttribute
    public void addIngredientsToModel(Model model) {
        Iterable<Ingredient> ingredients = ingredientRepo.findAll();
        Type[] types = Ingredient.Type.values();
        for (Type type : types) {
            model.addAttribute(type.toString().toLowerCase(),
                filterByType(ingredients, type));
        }
    }

    // ...
}
```

Метод addIngredientsToModel() извлекает все ингредиенты из базы данных, вызывая метод findAll() внедренного экземпляра IngredientRepository. Затем он фильтрует их по типам ингредиентов и добавляет в модель.

Теперь, реализовав IngredientRepository, с помощью которого можно получить объекты Ingredient, мы можем упростить IngredientByIdConverter, созданный в главе 2, заменив жестко определенный ассоциативный массив объектов Ingredient простым вызовом метода IngredientRepository.findById(), как показано в листинге 3.8.

Листинг 3.8 Упрощение IngredientByIdConverter

```
package tacos.web;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.convert.converter.Converter;
import org.springframework.stereotype.Component;

import tacos.Ingredient;
import tacos.data.IngredientRepository;

@Component
public class IngredientByIdConverter implements Converter<String, Ingredient> {

    private IngredientRepository ingredientRepo;

    @Autowired
    public IngredientByIdConverter(IngredientRepository ingredientRepo) {
        this.ingredientRepo = ingredientRepo;
    }

    @Override
    public Ingredient convert(String id) {
        return ingredientRepo.findById(id).orElse(null);
    }
}
```

Мы почти готовы запустить приложение и опробовать внесенные изменения. Но, прежде чем начать читать данные из таблицы `Ingredient`, на которую ссылаются запросы, нам следует создать эту таблицу и заполнить ее некоторыми данными.

3.1.3 Определение схемы и предварительная загрузка данных

Помимо таблицы `Ingredient`, нам также понадобятся таблицы для хранения информации о заказах и рецептах. Эти таблицы и отношения между ними показаны на рис. 3.1.

Таблицы на рис. 3.1 служат следующим целям:

- `Taco_Order` – хранит информацию о заказах;
- `Taco` – хранит рецепты тако;
- `Ingredient_Ref` – хранит одну или несколько записей для каждой записи в таблице `Taco`, представляющих ингредиенты для этого тако;
- `Ingredient` – хранит информацию об ингредиентах.

В нашем приложении тако не может существовать отдельно от заказа, соответственно, `Taco_Order` и `Taco` являются составляющими агрегата, где `Taco_Order` – корень этого агрегата. Объекты `Ingredient`, напротив, являются самостоятельными сущностями и образуют свой агрегат, и записи в `Taco` ссылаются на них посредством `Ingredient_Ref`.

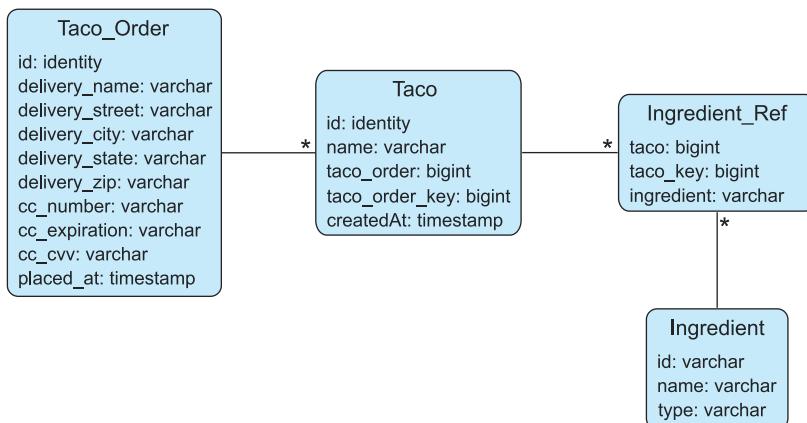


Рис. 3.1 Схема базы данных для приложения Taco Cloud

ПРИМЕЧАНИЕ Агрегаты и корни агрегатов – это базовые понятия *предметно-ориентированного проектирования (Domain-Driven Design, DDD)*; подхода к проектированию, продвигающего идею о том, что структура и имена сущностей в программном коде должны соответствовать предметной области. Мы используем некоторые идеи из предметно-ориентированного проектирования в объектах Taco Cloud, но вообще DDD – это гораздо больше, чем агрегаты и корни агрегатов. За дополнительной информацией по этому вопросу я советую обратиться к основополагающей работе Эрика Эванса (Eric Evans) «Domain-Driven Design: Tackling Complexity in the Heart of Software» (https://www.dddcommunity.org/book/evans_2003/)¹.

В листинге 3.9 показан SQL-код, создающий эти таблицы.

Листинг 3.9 Определение схемы базы данных для приложения Taco Cloud

```

create table if not exists Taco_Order (
    id identity,
    delivery_Name varchar(50) not null,
    delivery_Street varchar(50) not null,
    delivery_City varchar(50) not null,
    delivery_State varchar(2) not null,
    delivery_Zip varchar(10) not null,
    cc_number varchar(16) not null,
    cc_expiration varchar(5) not null,
    cc_cvv varchar(3) not null,
    placed_at timestamp not null

```

¹ Эванс Эрик. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем. Вильямс, 2020, ISBN: 978-5-6040724-9-3. – Прим. перев.

```

);
create table if not exists Taco (
    id identity,
    name varchar(50) not null,
    taco_order bigint not null,
    taco_order_key bigint not null,
    created_at timestamp not null
);
create table if not exists Ingredient_Ref (
    ingredient varchar(4) not null,
    taco bigint not null,
    taco_key bigint not null
);
create table if not exists Ingredient (
    id varchar(4) not null,
    name varchar(25) not null,
    type varchar(10) not null
);
alter table Taco
    add foreign key (taco_order) references Taco_Order(id);
alter table Ingredient_Ref
    add foreign key (ingredient) references Ingredient(id);

```

Но теперь возникает вопрос: куда поместить это определение схемы? Как оказывается, Spring Boot имеет ответ на этот вопрос.

Если в корне пути поиска классов приложения имеется файл с именем *schema.sql*, то его содержимое будет интерпретироваться как код на языке SQL и выполняться в базе данных при запуске приложения. Поэтому сохраните содержимое листинга 3.8 в файле с именем *schema.sql* в папке *src/main/resources*.

Нам также необходимо наполнить базу данных некоторыми начальными данными об ингредиентах. К счастью, Spring Boot тоже автоматически выполняет файл с именем *data.sql* из корня пути поиска классов при запуске приложения. То есть мы можем наполнить базу данных, используя операторы *insert* (листинг 3.10) в файле *src/main/resources/data.sql*.

Листинг 3.10 Заполнение базы данных начальными данными

```

delete from Ingredient_Ref;
delete from Taco;
delete from Taco_Order;

delete from Ingredient;
insert into Ingredient (id, name, type)
    values ('FLTO', 'Flour Tortilla', 'WRAP');
insert into Ingredient (id, name, type)
    values ('COTO', 'Corn Tortilla', 'WRAP');

```

```

insert into Ingredient (id, name, type)
    values ('GRBF', 'Ground Beef', 'PROTEIN');
insert into Ingredient (id, name, type)
    values ('CARN', 'Carnitas', 'PROTEIN');
insert into Ingredient (id, name, type)
    values ('TMT0', 'Diced Tomatoes', 'VEGGIES');
insert into Ingredient (id, name, type)
    values ('LETC', 'Lettuce', 'VEGGIES');
insert into Ingredient (id, name, type)
    values ('CHED', 'Cheddar', 'CHEESE');
insert into Ingredient (id, name, type)
    values ('JACK', 'Monterrey Jack', 'CHEESE');
insert into Ingredient (id, name, type)
    values ('SLSA', 'Salsa', 'SAUCE');
insert into Ingredient (id, name, type)
    values ('SRCR', 'Sour Cream', 'SAUCE');

```

Даже притом что мы определили репозиторий пока только для ингредиентов, мы уже можем запустить приложение Taco Cloud и открыть страницу создания рецепта, чтобы увидеть `JdbcIngredientRepository` в действии. Сделайте это! Когда вы вернетесь, мы напишем репозиторий для хранения Taco и `TacoOrder`.

3.1.4 Сохранение данных

Вы уже знаете, как использовать `JdbcTemplate` для сохранения объектов в базе данных. Мы написали метод `save()` в `JdbcIngredientRepository`, который вызывает метод `JdbcTemplate.update()`, чтобы сохранить объект `Ingredient` в базе данных.

Это был хороший пример, но, пожалуй, слишком простой. Как вы вскоре увидите, операция сохранения данных может выглядеть намного сложнее, чем в `JdbcIngredientRepository`.

В нашем проекте объекты `TacoOrder` и `Taco` являются частью агрегата, в котором `TacoOrder` – это корень агрегата. Иначе говоря, объекты `Taco` не могут существовать вне контекста `TacoOrder`. Итак, теперь нам нужно определить репозиторий для хранения объектов `TacoOrder` и заодно объектов `Taco`. Такой репозиторий можно определить в виде интерфейса `OrderRepository`:

```

package tacos.data;

import java.util.Optional;
import tacos.TacoOrder;

public interface OrderRepository {
    TacoOrder save(TacoOrder order);
}

```

Выглядит довольно просто, верно? Не совсем. Сохраняя `TacoOrder`, мы должны также сохранить связанные с ним объекты `Taco`. А сохра-

няя объекты Taco, мы должны сохранить объект, представляющий связь между Taco и каждым ингредиентом, входящим в рецепт тако. Связь между Taco и Ingredient определяет класс IngredientRef:

```
package tacos;

import lombok.Data;

@Data
public class IngredientRef {
    private final String ingredient;
}
```

Достаточно сказать, что на этот раз метод `save()` получится интереснее написанного выше соответствующего метода сохранения скромного объекта `Ingredient`.

Кроме всего прочего, метод `save()` должен определить, какой идентификатор присвоить заказу после сохранения. Согласно схеме, свойство `id` в таблице `Taco_Order` имеет тип `identity`, то есть база данных автоматически определяет его значение. Но если значение определяется базой данных, то как мы узнаем его, чтобы вернуть в объекте `TacoOrder`, возвращаемом из метода `save()`? К счастью, Spring предлагает полезный тип `GeneratedKeyHolder`, который может помочь в этом. Но это предполагает использование параметризованных запросов, как показано в следующей реализации метода `save()`:

```
package tacos.data;

import java.sql.Types;
import java.util.Arrays;
import java.util.Date;
import java.util.List;
import java.util.Optional;

import org.springframework.asm.Type;
import org.springframework.jdbc.core.JdbcOperations;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.core.PreparedStatementCreatorFactory;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import tacos.IngredientRef;
import tacos.Taco;
import tacos.TacoOrder;

@Repository
public class JdbcOrderRepository implements OrderRepository {

    private JdbcOperations jdbcOperations;

    public JdbcOrderRepository(JdbcOperations jdbcOperations) {
```

```

        this.jdbcOperations = jdbcOperations;
    }

    @Override
    @Transactional
    public TacoOrder save(TacoOrder order) {
        PreparedStatementCreatorFactory pscf =
            new PreparedStatementCreatorFactory(
                "insert into Taco_Order "
                + "(delivery_name, delivery_street, delivery_city, "
                + "delivery_state, delivery_zip, cc_number, "
                + "cc_expiration, cc_cvv, placed_at) "
                + "values (?, ?, ?, ?, ?, ?, ?, ?, ?)",
                Types.VARCHAR, Types.VARCHAR, Types.VARCHAR,
                Types.VARCHAR, Types.VARCHAR, Types.VARCHAR,
                Types.VARCHAR, Types.VARCHAR, Types.TIMESTAMP
            );
        pscf.setReturnGeneratedKeys(true);

        order.setPlacedAt(new Date());
        PreparedStatementCreator psc =
            pscf.newPreparedStatementCreator(
                Arrays.asList(
                    order.getDeliveryName(),
                    order.getDeliveryStreet(),
                    order.getDeliveryCity(),
                    order.getDeliveryState(),
                    order.getDeliveryZip(),
                    order.getCcNumber(),
                    order.getCcExpiration(),
                    order.getCcCVV(),
                    order.getPlacedAt())));
        GeneratedKeyHolder keyHolder = new GeneratedKeyHolder();
        jdbcOperations.update(psc, keyHolder);
        long orderId = keyHolder.getKey().longValue();
        order.setId(orderId);

        List<Taco> tacos = order.getTacos();
        int i=0;
        for (Taco taco : tacos) {
            saveTaco(orderId, i++, taco);
        }
        return order;
    }
}

```

На первый взгляд, в методе `save()` выполняется множество операций, однако их можно сгруппировать в несколько важных шагов. Первый шаг – создание объекта `PreparedStatementCreatorFactory`, описывающего запрос `insert` вместе с типами параметров запроса. Поскольку позднее нам понадобится идентификатор сохраненного

заказа, мы также должны вызвать метод `setReturnGeneratedKeys(true)` этого объекта.

После создания `PreparedStatementCreatorFactory` мы используем его для создания объекта `PreparedStatementCreator`, передавая значения из объекта `TacoOrder`, которые требуется сохранить. Последний аргумент в вызове `PreparedStatementCreator` – это дата создания заказа, которую также нужно будет установить в самом объекте `TacoOrder`, чтобы возвращаемый экземпляр `TacoOrder` содержал эту информацию.

После создания `PreparedStatementCreator` можно фактически сохранить заказ, вызвав метод `JdbcTemplate.update()`, передав ему `PreparedStatementCreator` и `GeneratedKeyHolder`. После сохранения заказа в `GeneratedKeyHolder` будет храниться значение поля `id`, назначенное базой данных, которое затем следует скопировать в свойство `id` объекта `TacoOrder`.

Теперь, после сохранения заказа, необходимо также сохранить объекты `Taco`, связанные с заказом. Это можно сделать вызовом метода `saveTaco()` для каждого `Taco` в заказе.

Метод `saveTaco()` очень похож на метод `save()`:

```
private long saveTaco(Long orderId, int orderKey, Taco taco) {
    taco.setCreatedAt(new Date());
    PreparedStatementCreatorFactory pscf =
        new PreparedStatementCreatorFactory(
            "insert into Taco "
            + "(name, created_at, taco_order, taco_order_key) "
            + "values (?, ?, ?, ?)",
            Types.VARCHAR, Types.TIMESTAMP, Type.LONG, Type.LONG
        );
    pscf.setReturnGeneratedKeys(true);

    PreparedStatementCreator psc =
        pscf.newPreparedStatementCreator(
            Arrays.asList(
                taco.getName(),
                taco.getCreatedAt(),
                orderId,
                orderKey));
}

GeneratedKeyHolder keyHolder = new GeneratedKeyHolder();
jdbcOperations.update(psc, keyHolder);
long tacoId = keyHolder.getKey().longValue();
taco.setId(tacoId);

saveIngredientRefs(tacoId, taco.getIngredients());

return tacoId;
}
```

Метод `saveTaco()` копирует структуру `save()`, но сохраняет объект `Taco`, а не `TacoOrder`. В конце он вызывает `saveIngredientRefs()`

для создания записи в таблице `Ingredient_Ref`, чтобы связать запись в таблице `Taco` с записью в таблице `Ingredient`. Ниже приводится реализация метода `saveIngredientRefs()`:

```
private void saveIngredientRefs(
    long tacoId, List<IngredientRef> ingredientRefs) {
    int key = 0;
    for (IngredientRef ingredientRef : ingredientRefs) {
        jdbcOperations.update(
            "insert into Ingredient_Ref (ingredient, taco, taco_key) "
            + "values (?, ?, ?)",
            ingredientRef.getIngredient(), tacoId, key++);
    }
}
```

Метод `saveIngredientRefs()` намного проще. Он циклически перебирает список объектов `Ingredient` и сохраняет каждый в таблице `Ingredient_Ref`. Он также имеет локальную переменную `key`, которая используется в качестве индекса, гарантирующего сохранность порядка следования ингредиентов.

Теперь, когда `OrderRepository` готов, его нужно внедрить в `OrderController` и использовать для сохранения заказа. В листинге 3.11 показаны изменения, которые необходимо внести для внедрения репозитория.

Листинг 3.11 Внедрение и использование репозитория OrderRepository

```
package tacos.web;
import javax.validation.Valid;

import org.springframework.stereotype.Controller;
import org.springframework.validation.Errors;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

import tacos.TacoOrder;
import tacos.data.OrderRepository;

@Controller
@RequestMapping("/orders")
@SessionAttributes("tacoOrder")
public class OrderController {

    private OrderRepository orderRepo;

    public OrderController(OrderRepository orderRepo) {
        this.orderRepo = orderRepo;
    }
}
```

```
// ...

@PostMapping
public String processOrder(@Valid TacoOrder order, Errors errors,
    SessionStatus sessionStatus) {
    if (errors.hasErrors()) {
        return "orderForm";
    }
    orderRepo.save(order);
    sessionStatus.setComplete();

    return "redirect:/";
}

}
```

Теперь конструктор принимает параметр `OrderRepository` и сохраняет его в переменной экземпляра, которая затем будет использоваться в методе `processOrder()`. В сам метод `processOrder()` добавился вызов метода `OrderRepository.save()`, заменивший операцию записи объекта `TacoOrder` в журнал.

Класс `JdbcTemplate` в Spring значительно упрощает работу с реляционными базами данных. Но даже при использовании `JdbcTemplate` некоторые задачи сохранения остаются довольно сложными в реализации, особенно когда требуется организовать сохранение вложенных объектов. А нет ли еще более простого способа работы с JDBC?

Давайте познакомимся с проектом Spring Data JDBC, упрощающим работу с JDBC еще больше, даже когда требуется сохранять агрегаты.

3.2 Spring Data JDBC

Проект Spring Data – это довольно большой зонтичный проект, включающий несколько подпроектов, большинство из которых сосредоточены на хранении данных в различных типах баз данных. Вот некоторые из самых популярных проектов Spring Data:

- *Spring Data JDBC* – поддержка интерфейса JDBC для реляционных баз данных;
- *Spring Data JPA* – поддержка интерфейса JPA для реляционных баз данных;
- *Spring Data MongoDB* – поддержка документной базы данных Mongo;
- *Spring Data Neo4j* – поддержка графовой базы данных Neo4j;
- *Spring Data Redis* – поддержка хранилища ключей Redis;
- *Spring Data Cassandra* – поддержка колоночной базы данных Cassandra.

Одной из самых интересных и полезных функций, общей для всех проектов в Spring Data, является возможность автоматического создания репозиториев на основе интерфейса спецификации. Как резуль-

тат поддержка хранения данных в проектах, использующих Spring Data, практически не содержит логики для работы с хранилищами и включает лишь один или несколько интерфейсов спецификаций.

Давайте посмотрим, как можно использовать Spring Data JDBC в нашем проекте, чтобы упростить хранение данных с помощью JDBC. Прежде всего нам нужно добавить зависимость Spring Data JDBC в спецификацию сборки проекта.

3.2.1 Добавление Spring Data JDBC в спецификацию сборки

Spring Data JDBC доступен приложениям Spring Boot в виде начальной зависимости `spring-boot-starter-data-jdbc`. Вот как она выглядит в файле `pom.xml` (листинг 3.12):

Листинг 3.12 Добавление начальной зависимости Spring Data JDBC в спецификацию сборки

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
```

Нам больше не понадобится начальная зависимость, благодаря которой мы получили `JdbcTemplate`, поэтому удалим следующие строки:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

Однако нам все еще нужна база данных, поэтому оставим на месте зависимость для H2.

3.2.2 Определение интерфейсов репозиториев

К счастью, мы уже создали `IngredientRepository` и `OrderRepository`, поэтому можно считать, что большая часть работы уже выполнена. Но нам нужно немного изменить их, чтобы использовать совместно с Spring Data JDBC.

Spring Data может автоматически генерировать реализации для интерфейсов репозиториев, но только если эти интерфейсы наследуют стандартные интерфейсы, предоставляемые Spring Data. В данном случае наши интерфейсы репозиториев должны наследовать интерфейс `Repository`, чтобы библиотека Spring Data смогла создать реализацию автоматически. Например, вот как можно добавить наследование `Repository` в определение `IngredientRepository`:

```
package tacos.data;
import java.util.Optional;
```

```

import org.springframework.data.repository.Repository;
import tacos.Ingredient;

public interface IngredientRepository
    extends Repository<Ingredient, String> {

    Iterable<Ingredient> findAll();

    Optional<Ingredient> findById(String id);

    Ingredient save(Ingredient ingredient);

}

```

Как видите, интерфейс `Repository` параметризован. Первый параметр – это тип объектов, которые будут храниться в репозитории, в данном случае `Ingredient`. Второй параметр – это тип поля идентификатора хранимого объекта. Для объектов `Ingredient` это `String`.

Помимо `Repository`, наследования которого вполне достаточно для нормальной работы `IngredientRepository`, Spring Data предлагает также интерфейс `CrudRepository`, реализующий, кроме всего прочего, три метода, которые мы определили в `IngredientRepository`. То есть часто удобнее вместо `Repository` унаследовать `CrudRepository`, как показано в листинге 3.13.

Листинг 3.13 Определение репозитория для хранения ингредиентов

```

package tacos.data;

import org.springframework.data.repository.CrudRepository;
import tacos.Ingredient;

public interface IngredientRepository
    extends CrudRepository<Ingredient, String> {
}

```

Аналогично можно определить `OrderRepository`, наследующий `CrudRepository`, как показано в листинге 3.14.

Листинг 3.14 Определение репозитория для хранения заказов

```

package tacos.data;

import org.springframework.data.repository.CrudRepository;
import tacos.TacoOrder;

public interface OrderRepository
    extends CrudRepository<TacoOrder, Long> {
}

```

Интерфейс `CrudRepository` уже определяет нужные нам методы, поэтому в обоих случаях нет необходимости явно определять их в интерфейсах `IngredientRepository` и `OrderRepository`.

Теперь у нас есть два репозитория. Вы могли бы подумать, что далее нам предстоит написать реализации их обоих, включая дюжину методов, определяемых интерфейсом `CrudRepository`. Но спешу вас обрадовать, Spring Data предоставляет готовые реализации этих методов! Когда приложение запускается, Spring Data автоматически создает необходимые реализации на лету. Это означает, что репозитории немедленно готовы к использованию. Просто внедрите их в контроллеры, и все готово.

Более того, поскольку Spring Data автоматически создает реализации интерфейсов во время выполнения, вам больше не нужны явные реализации в `JdbcIngredientRepository` и `JdbcOrderRepository`. Вы можете удалить эти два класса и навсегда забыть про них!

3.2.3 Аннотирование классов данных предметной области

Далее нам нужно аннотировать классы данных предметной области, чтобы библиотека Spring Data JDBC знала, как их сохранять. Фактически аннотирование сводится к добавлению аннотаций `@Id` к свойствам, определяющим идентичность, чтобы Spring Data знала, какое поле уникально идентифицирует объекты, и, возможно, аннотаций `@Table` к классам.

В листинге 3.15 показано, как добавить аннотации `@Table` и `@Id` в класс `TacoOrder`.

Листинг 3.15 Подготовка класса TacoOrder к сохранению в базе данных

```
package tacos;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.validation.constraints.Digits;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Pattern;

import org.hibernate.validator.constraints.CreditCardNumber;
import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Table;

import lombok.Data;

@Data
@Table
public class TacoOrder implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
```

```
private Long id;
// ...
}
```

Аннотация `@Table` не является обязательной. По умолчанию для хранения объектов класса создается таблица с именем, соответствующим имени этого класса. В данном случае для хранения объектов `TacoOrder` будет создана таблица с именем `"Taco_Order"`. Если вас это устраивает, то можете просто убрать аннотацию `@Table` или использовать ее без параметров. Но если вы решите использовать другое имя таблицы, то укажите его в параметре аннотации `@Table`, как показано ниже:

```
@Table("Taco_Cloud_Order")
public class TacoOrder {
    ...
}
```

В данном случае объекты `TacoOrder` будут сохраняться в таблице с именем `"Taco_Cloud_Order"`.

Аннотация `@Id` определяет свойство `id` как уникальный идентификатор объектов `TacoOrder`. Все остальные свойства в `TacoOrder` будут автоматически отображаться в столбцы с именами, соответствующими именам этих свойств. Например, свойство `deliveryName` будет автоматически сохраняться в столбце с именем `delivery_name`. Но если вы решите явно определить имя столбца для того или иного свойства, то добавьте к нему аннотацию `@Column`, как показано ниже:

```
@Column("customer_name")
@NotBlank(message="Delivery name is required")
private String deliveryName;
```

В данном случае аннотация `@Column` указывает, что свойство `deliveryName` должно сохраняться в столбце с именем `customer_name`.

Аннотации `@Table` и `@Id` необходимо применить также к другим классам данных предметной области, включая `Ingredient` (листинг 3.16).

Листинг 3.16 Подготовка класса Ingredient к сохранению в базе данных

```
package tacos;

import org.springframework.data.annotation.Id;
import org.springframework.data.domain.Persistable;
import org.springframework.data.relational.core.mapping.Table;

import lombok.AccessLevel;
import lombok.AllArgsConstructor;
```

```

import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Table
@AllArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
public class Ingredient implements Persistable<String> {

    @Id
    private String id;

    // ...
}

```

... и Taco (листинг 3.17).

Листинг 3.17 Подготовка класса Taco к сохранению в базе данных

```

package tacos;

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

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.springframework.data.annotation.Id;
import org.springframework.data.relational.core.mapping.Table;

import lombok.Data;

@Data
@Table
public class Taco {

    @Id
    private Long id;

    // ...
}

```

Объекты IngredientRef автоматически будут сохраняться в таблице с именем "Ingredient_Ref", что идеально подходит для нашего приложения. Этот класс можно было бы снабдить аннотацией @Table, но в данном случае в этом нет необходимости. В таблице "Ingredient_Ref" нет столбца типа identity, поэтому никакие свойства в классе IngredientRef не нужно аннотировать с помощью @Id.

После внесения этих небольших изменений, в добавок к полному удалению классов JdbcIngredientRepository и JdbcOrderRepository, у нас осталось намного меньше кода, имеющего отношение к базам

данных. Несмотря на это, реализации репозиториев, созданные библиотекой Spring Data во время выполнения, делают все то же самое, что и репозитории, реализованные с использованием `JdbcTemplate`. На самом деле автоматически генерированные репозитории способны даже на большее, потому что два наших интерфейса репозиториев наследуют `CrudRepository`, который автоматически реализует около дюжины операций для создания, чтения, изменения и удаления объектов.

3.2.4 Предварительная загрузка данных с помощью `CommandLineRunner`

При работе с `JdbcTemplate` мы организовали предварительную загрузку объектов `Ingredient` на запуске приложения, создав файл `data.sql`, который использовался для заполнения базы данных в момент создания bean-компоненты, представляющей источник данных. Тот же подход можно использовать с Spring Data JDBC. На самом деле он будет работать с любым механизмом хранения данных, основанным на реляционной базе данных. Но давайте рассмотрим другой способ предварительного заполнения базы данных, который предлагает немного больше гибкости.

В Spring Boot имеются два полезных интерфейса для выполнения логики при запуске приложения: `CommandLineRunner` и `ApplicationRunner`. Эти два интерфейса очень похожи. Оба являются функциональными интерфейсами, требующими реализации одного метода `run()`. На этапе запуска, после включения всех компонентов в контекст приложения Spring, но до того, как произойдет что-либо еще, для всех компонентов в контексте, реализующих интерфейс `CommandLineRunner` или `ApplicationRunner`, будут вызываться их методы `run()`. Это удобный момент для загрузки начальных данных в базу данных.

Поскольку `CommandLineRunner` и `ApplicationRunner` являются функциональными интерфейсами, их легко объявить как bean-компоненты в классе конфигурации с помощью метода с аннотацией `@Bean`, который возвращает лямбда-функцию. Например, вот как можно создать bean-компонент `CommandLineRunner` для загрузки данных:

```
@Bean
public CommandLineRunner dataLoader(IngredientRepository repo) {
    return args -> {
        repo.save(new Ingredient("FLTO", "Flour Tortilla", Type.WRAP));
        repo.save(new Ingredient("COTO", "Corn Tortilla", Type.WRAP));
        repo.save(new Ingredient("GRBF", "Ground Beef", Type.PROTEIN));
        repo.save(new Ingredient("CARN", "Carnitas", Type.PROTEIN));
        repo.save(new Ingredient("TMT0", "Diced Tomatoes", Type.VEGGIES));
        repo.save(new Ingredient("LETC", "Lettuce", Type.VEGGIES));
        repo.save(new Ingredient("CHED", "Cheddar", Type.CHEESE));
        repo.save(new Ingredient("JACK", "Monterrey Jack", Type.CHEESE));
```

```

repo.save(new Ingredient("SLSA", "Salsa", Type.SAUCE));
repo.save(new Ingredient("SRCR", "Sour Cream", Type.SAUCE));
};

}

```

Здесь `IngredientRepository` внедряется в метод `bean`-компонента и используется в лямбда-выражении для создания объектов `Ingredient`. Метод `run()` интерфейса `CommandLineRunner` принимает единственный параметр – строку со всеми аргументами командной строки для приложения. В данном случае они не нужны для загрузки ингредиентов в базу данных, поэтому параметр `args` игнорируется.

Мы могли бы также определить `bean`-компонент для загрузки данных как лямбда-реализацию `ApplicationRunner`:

```

@Bean
public ApplicationRunner dataLoader(IngredientRepository repo) {
    return args -> {
        repo.save(new Ingredient("FLTO", "Flour Tortilla", Type.WRAP));
        repo.save(new Ingredient("COTO", "Corn Tortilla", Type.WRAP));
        repo.save(new Ingredient("GRBF", "Ground Beef", Type.PROTEIN));
        repo.save(new Ingredient("CARN", "Carnitas", Type.PROTEIN));
        repo.save(new Ingredient("TMTO", "Diced Tomatoes", Type.VEGGIES));
        repo.save(new Ingredient("LETC", "Lettuce", Type.VEGGIES));
        repo.save(new Ingredient("CHED", "Cheddar", Type.CHEESE));
        repo.save(new Ingredient("JACK", "Monterrey Jack", Type.CHEESE));
        repo.save(new Ingredient("SLSA", "Salsa", Type.SAUCE));
        repo.save(new Ingredient("SRCR", "Sour Cream", Type.SAUCE));
    };
}

```

Ключевое различие между `CommandLineRunner` и `ApplicationRunner` заключается в параметре метода `run()`. В `CommandLineRunner` этот метод принимает строку со всеми параметрами командной строки, а `ApplicationRunner` – параметр типа `ApplicationArguments`, предлагающий методы для доступа к отдельным аргументам, извлеченным из командной строки.

Например, предположим, что мы решили добавить в приложение поддержку аргументов командной строки, таких как `--version 1.2.3`, и анализировать их в нашем компоненте-загрузчике. При использовании `CommandLineRunner` нам пришлось бы отыскать в массиве подстроку `--version`, а затем выбрать следующее за ней значение. `ApplicationRunner`, напротив, позволяет прямо запросить из `ApplicationArguments` значение для аргумента `--version`:

```

public ApplicationRunner dataLoader(IngredientRepository repo) {
    return args -> {
        List<String> version = args.getOptionValues("version");
        ...
    };
}

```

Метод `getOptionValues()` возвращает `List<String>`, что позволяет несколько раз передать один и тот же параметр командной строки.

Однако нашему приложению не нужны аргументы командной строки для загрузки данных. Поэтому наш компонент загрузки данных просто игнорирует параметр `args`.

Удобство применения интерфейса `CommandLineRunner` или `ApplicationRunner` для первоначальной загрузки данных состоит в том, что в этом случае вместо SQL-сценария используются репозитории, то есть они одинаково хорошо будут работать и с реляционными, и с нереляционными базами данных. Эта особенность пригодится нам в следующей главе, когда мы будем рассматривать вариант использования Spring Data для хранения данных в нереляционных базах данных.

Но прежде давайте познакомимся с еще одним проектом Spring Data, реализующим поддержку хранения данных в реляционных базах данных: Spring Data JPA.

3.3 Хранение данных с помощью *Spring Data JPA*

Помимо Spring Data JDBC, упрощающего организацию хранения данных, существует еще один популярный проект, обеспечивающий поддержку реляционных баз данных, – Java Persistence API (JPA). Spring Data JPA предлагает подход к работе с базами данных с применением JPA, подобно тому, как Spring Data JDBC предлагает подход с применением JDBC.

Чтобы опробовать этот проект, мы сначала заменим репозитории на основе JDBC, созданные выше в этой главе, репозиториями Spring Data JPA, но перед этим добавим зависимость Spring Data JPA в спецификацию сборки приложения.

3.3.1 Добавление зависимости *Spring Data JPA*

Зависимость Spring Data JPA добавляется в приложения Spring Boot с помощью начальной зависимости JPA. Она подключает саму библиотеку Spring Data JPA и, транзитивно, библиотеку Hibernate, реализующую механизм JPA:

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

Если вам понадобится использовать другую реализацию JPA, то исключите зависимость Hibernate и подключите нужную библиотеку JPA. Например, вот как можно задействовать EclipseLink вместо Hibernate:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.hibernate</groupId>
            <artifactId>hibernate-core</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.eclipse.persistence</groupId>
    <artifactId>org.eclipse.persistence.jpa</artifactId>
    <version>2.7.6</version>
</dependency>
```

Обратите внимание, что в зависимости от выбранной реализации JPA вам может потребоваться внести другие изменения. За более подробной информацией обращайтесь к документации с описанием выбранной реализации JPA. А теперь вернемся к объектам предметной области и добавим к ним аннотации поддержки JPA.

3.3.2 Аннотирование классов данных предметной области

Как вы уже видели на примере Spring Data JDBC, Spring Data предлагает удивительные возможности, когда дело доходит до создания репозиториев. Но, к сожалению, это мало помогает в случае с аннотированием объектов данных предметной области аннотациями JPA. Нам нужно открыть классы `Ingredient`, `Taco` и `TacoOrder` и добавить несколько аннотаций. Сначала рассмотрим класс `Ingredient` (листинг 3.18).

Листинг 3.18 Аннотирование класса `Ingredient` для поддержки JPA

```
package tacos;

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

import lombok.AccessLevel;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Entity
@AllArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
public class Ingredient {

    @Id
```

```

private String id;
private String name;
private Type type;

public enum Type {
    WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
}

}

```

Чтобы объявить класс `Ingredient` сущностью JPA, его нужно снабдить аннотацией `@Entity`, а свойство `id` следует отметить аннотацией `@Id`, чтобы обозначить его как свойство, однозначно идентифицирующее объект в базе данных. Обратите внимание, что эта аннотация `@Id` реализована в пакете `javax.persistence`, а не в `org.springframework.data.annotation`, как аннотация `@Id`, предоставляемая проектом Spring Data.

Также отметьте, что нам больше не нужна ни аннотация `@Table`, ни реализация интерфейса `Persistable`. Мы могли бы продолжать использовать аннотацию `@Table`, но при применении JPA в этом нет необходимости, и по умолчанию в качестве имени таблицы используется имя класса (в данном случае `Ingredient`). Интерфейс `Persistable` был нужен при работе с Spring Data JDBC, только чтобы определить, должен ли создаваться новый или обновляться существующий объект; JPA определяет это автоматически.

Как можно заметить, кроме аннотаций JPA, мы добавили также аннотацию `@NoArgsConstructor` на уровне класса. JPA требует, чтобы сущности имели конструктор без аргументов, и аннотация `@NoArgsConstructor` из библиотеки Lombok автоматически создает такой конструктор. Однако мы не собираемся давать возможность использовать его извне, поэтому объявили его приватным, передав атрибут `access` со значением `AccessLevel.PRIVATE`. Мы также должны сделать все свойства финальными, поэтому добавили атрибут `force` со значением `true`, в результате чего конструктор, сгенерированный библиотекой Lombok, присвоит им значение по умолчанию `null`, `0` или `false`, в зависимости от типа свойства.

Мы добавили аннотацию `@AllArgsConstructor`, чтобы упростить создание объекта `Ingredient` со всеми инициализированными свойствами.

Нам также понадобится `@RequiredArgsConstructor`. Аннотация `@Data` неявно добавляет конструктор с обязательными аргументами, но, когда используется `@NoArgsConstructor`, этот конструктор удаляется. Явное добавление аннотации `@RequiredArgsConstructor` гарантирует, что мы по-прежнему будем иметь конструктор со всеми обязательными аргументами, помимо приватного конструктора без аргументов.

Теперь перейдем к классу `Taco` и посмотрим, как превратить его в сущность JPA (листинг 3.19).

Листинг 3.19 Аннотирование класса Taco для поддержки JPA

```
package tacos;

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

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

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import lombok.Data;

@Entity
@Data
public class Taco {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @NotNull
    @Size(min=5, message="Name must be at least 5 characters long")
    private String name;

    private Date createdAt = new Date();

    @Size(min=1, message="You must choose at least 1 ingredient")
    @ManyToOne()
    private List<Ingredient> ingredients = new ArrayList<>();

    public void addIngredient(Ingredient ingredient) {
        this.ingredients.add(ingredient);
    }

}
```

Класс Taco, так же как Ingredient, теперь снабжен аннотацией `@Entity`, а его свойство `id` – аннотацией `@Id`. В отношении создания уникальных значений идентификатора мы полагаемся на базу данных, поэтому добавили к свойству `id` еще и аннотацию `@GeneratedValue`, передав ей атрибут `strategy` со значением `AUTO`.

Чтобы объявить взаимосвязь между Taco и списком ингредиентов Ingredient, мы отметили список `ingredients` аннотацией `@ManyToOne`. Объект Taco может включать в список несколько объектов Ingredient, а один объект Ingredient может быть частью списков в нескольких объектах Taco.

Наконец, аннотируем класс TacoOrder. В листинге 3.20 показан новый класс TacoOrder.

Листинг 3.20 Аннотирование класса TacoOrder для поддержки JPA

```
package tacos;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.persistence.CascadeType;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;

import javax.validation.constraints.Digits;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Pattern;

import org.hibernate.validator.constraints.CreditCardNumber;

import lombok.Data;

@Data
@Entity
public class TacoOrder implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private Date placedAt = new Date();

    ...

    @OneToMany(cascade = CascadeType.ALL)
    private List<Taco> tacos = new ArrayList<>();

    public void addTaco(Taco taco) {
        this.tacos.add(taco);
    }

}
```

Как видите, изменения в TacoOrder очень похожи на изменения в Taco. Стоит отметить одну важную деталь: связь со списком объектов Taco аннотирована с помощью @OneToMany, то есть все тачо в этом списке относятся к этому одному заказу. Кроме того, анно-

тации передается атрибут `cascade` со значением `CascadeType.ALL`, поэтому при удалении заказа все связанные с ним также будут удалены.

3.3.3 Объявление репозиториев JPA

Создавая репозитории `JdbcTemplate`, мы явно объявляли методы, которые должны были иметь эти репозитории. Переключившись на Spring Data JDBC, мы смогли отказаться от явных классов реализаций, просто унаследовав интерфейс `CrudRepository`. Как оказалось, `CrudRepository` можно также использовать и с Spring Data JPA. Например, вот новый интерфейс `IngredientRepository`:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Ingredient;

public interface IngredientRepository
    extends CrudRepository<Ingredient, String> { }
```

На самом деле интерфейс `IngredientRepository`, который мы будем использовать с Spring Data JPA, идентичен использовавшемуся с Spring Data JDBC. Интерфейс `CrudRepository` часто используется в проектах Spring Data независимо от механизма хранения. Точно также можно определить `OrderRepository`:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;
import tacos.TacoOrder;

public interface OrderRepository
    extends CrudRepository<TacoOrder, Long> { }
```

Методы, предоставляемые интерфейсом `CrudRepository`, отлично подходят для организации хранения обычных сущностей. Но как быть, если имеются какие-то особые требования? Давайте посмотрим, как настроить репозитории для выполнения запросов, уникальных для вашей предметной области.

3.3.4 Специализация репозиториев

Представьте, что кроме основных операций CRUD (Create, Read, Update, Delete – создать, прочитать, изменить, удалить), предоставляемых интерфейсом `CrudRepository`, необходимо также иметь возможность получить список всех заказов, доставленных в заданный район.

Как оказывается, эту задачу легко решить, добавив следующее объявление метода в `OrderRepository`:

```
List<TacoOrder> findByDeliveryZip(String deliveryZip);
```

Генерируя реализацию репозитория, Spring Data проверяет все методы в его интерфейсе, анализирует их имена и пытается определить назначение каждого метода в контексте хранимого объекта (в данном случае `TacoOrder`). По сути, Spring Data определяет своеобразный миниатюрный предметно-ориентированный язык (Domain-Specific Language, DSL), в котором детали операций с базой данных выражаются в сигнатурах методов.

Spring Data понимает, что метод `findByDeliveryZip()` предназначен для поиска заказов, потому что возвращаемое значение параметризовано типом `TacoOrder`. Имя метода `findByDeliveryZip()` ясно дает понять, что он должен найти все сущности `TacoOrder`, сопоставив свойство `deliveryZip` каждой из них со значением, переданным в аргументе.

Метод `findByDeliveryZip()` достаточно прост, но Spring Data может обрабатывать и более сложные имена методов. Методы репозитория формируются из глагола, необязательного подлежащего, слова `By` и предиката. В имени `findByDeliveryZip()` глаголом является `find` (найти), а предикатом – `DeliveryZip` (почтовый индекс, или район доставки); подлежащее не указано, и по умолчанию подразумевается `TacoOrder`.

Рассмотрим другой, более сложный пример. Предположим, нам нужно запросить все заказы, доставленные в определенный район в определенном диапазоне дат. В этом случае может оказаться полезным следующий метод в `OrderRepository`:

```
List<TacoOrder> readOrdersByDeliveryZipAndPlacedAtBetween(
    String deliveryZip, Date startDate, Date endDate);
```

На рис. 3.2 показано, как Spring Data анализирует имя метода `readOrdersByDeliveryZipAndPlacedAtBetween()` при создании реализации репозитория. Как видите, глаголом в данном случае является `read` (прочитать). Spring Data воспринимает глаголы `find` (найти), `read` (прочитать) и `get` (получить) как синонимы, обозначающие выборку одного или нескольких объектов. Кроме того, в качестве глагола можно также использовать `count` (подсчитать), если нужно, чтобы метод вернул только целое число с количеством сущностей, соответствующих предикату.

Подлежащее в имени метода может отсутствовать, но в данном случае оно указано – это `Orders` (заказы). Spring Data игнорирует большинство слов, служащих подлежащим, поэтому с таким же успехом методу можно было дать имя `readPuppiesBy...`, и он все равно отыскал бы сущности `TacoOrder`, потому что этим типом параметризован `CrudRepository`.

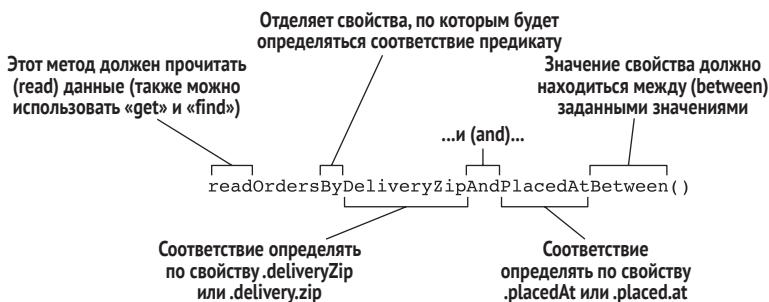


Рис. 3.2 Spring Data анализирует сигнатуру метода репозитория, чтобы определить, какой запрос выполнить

Предикат следует за словом By в имени метода и является самой интересной частью сигнатуры метода. В данном случае предикат ссылается на два свойства класса TacoOrder: deliveryZip и PlacedAt. Свойство deliveryZip должно быть равно значению, переданному в первом параметре. Ключевое слово Between указывает, что значение свойства PlacedAt должно находиться между значениями, переданными в двух последних параметрах.

В дополнение к неявной операции Equals (равно) и операции Between (между) сигнатуры методов репозиториев Spring Data могут включать любой из следующих операторов:

- IsAfter, After, IsGreaterThan, GreaterThan;
- IsGreaterThanOrEqual, GreaterThanOrEqual;
- IsBefore, Before, IsLessThan, LessThan;
- IsLessThanOrEqual, LessThanOrEqual;
- IsBetween, Between;
- IsNull, Null;
- IsNotNull, NotNull;
- IsIn, In;
- IsNotIn, NotIn;
- IsStartingWith, StartingWith, StartsWith;
- IsEndingWith, EndingWith, EndsWith;
- IsContaining, Containing, Contains;
- IsLike, Like;
- IsNotLike, NotLike;
- IsTrue, True;
- IsFalse, False;
- Is, Equals;
- IsNot, Not;
- IgnoringCase, IgnoresCase.

Как вариант вместо IgnoringCase и IgnoresCase можно использовать AllIgnoringCase или AllIgnoresCase, чтобы показать, что при со-поставлении строк регистр символов не должен учитываться. Например, взгляните на имя следующего метода:

```
List<TacoOrder> findByDeliveryToAndDeliveryCityAllIgnoresCase(
    String deliveryTo, String deliveryCity);
```

Наконец, в конце имени метода можно добавить `OrderBy`, чтобы отсортировать результаты по указанному столбцу. Например, вот как можно выполнить сортировку результатов по свойству `deliveryTo`:

```
List<TacoOrder> findByDeliveryCityOrderByDeliveryTo(String city);
```

Такое соглашение об именах может быть очень удобным для относительно простых запросов, но не нужно обладать большим воображением, чтобы заметить, что для более сложных запросов имена методов могут получиться весьма неудобоваримыми. В подобных случаях просто давайте методам имена по своему вкусу и аннотируйте их аннотацией `@Query` с явно указанным запросом, который должен выполняться при вызове метода, как показано ниже:

```
@Query("Order o where o.deliveryCity='Seattle'")
List<TacoOrder> readOrdersDeliveredInSeattle();
```

В этом простом примере запрос в аннотации `@Query` выберет все заказы, доставленные в Сиэтл. Но вообще в `@Query` можно определить практически любой запрос JPA, который только можно придумать, но трудно или невозможно выразить в виде имени метода.

Специализированные методы запросов также можно использовать совместно с Spring Data JDBC, но в этом случае необходимо учитывать следующие важные отличия:

- все специализированные методы запросов должны снабжаться аннотацией `@Query`. Это связано с тем, что, в отличие от JPA, Spring Data JDBC не поддерживает анализ имен методов и не может автоматически конструировать запросы;
- все запросы, указанные в `@Query`, должны быть SQL-запросами, а не запросами JPA.

В следующей главе мы рассмотрим возможность использования Spring Data для работы с нереляционными базами данных. Там вы увидите, что специализированные методы запросов работают очень похоже, хотя язык запросов в `@Query` будет отличаться, в зависимости от используемой базы данных.

Итоги

- `JdbcTemplate` значительно упрощает работу с JDBC.
- `PreparedStatementCreator` и `KeyHolder` можно использовать вместе, когда нужно узнать значение идентификатора, генерированного базой данных.
- Spring Data JDBC и Spring Data JPA упрощают работу с реляционными базами данных, сводя ее к определению интерфейсов репозиториев.

Работа с нереляционными данными

В этой главе рассматриваются следующие темы:

- хранение данных в Cassandra;
- моделирование данных в Cassandra;
- работа с документными данными в MongoDB.

Говорят, что разнообразие делает жизнь насыщеннее.

Вероятно, у вас есть любимый сорт мороженого, вкус которого вам нравится больше всего, потому что он полнее удовлетворяет желание насладиться этим сливочным чудом. Но многие из нас, несмотря на личные предпочтения, время от времени пробуют разные вкусы, чтобы попробовать что-то новое, ощутить разницу.

Базы данных – это своего рода мороженое. На протяжении десятилетий излюбленным средством хранения данных были реляционные базы данных. Но в настоящее время мы имеем намного больше возможностей, чем когда-либо прежде. Так называемые базы данных «NoSQL» (<https://aws.amazon.com/nosql/>) предлагают различные концепции и структуры для хранения данных. И хотя в большинстве случаев выбор по-прежнему определяется вкусом, некоторые базы данных лучше подходят для хранения данных определенных типов, чем другие.

К счастью, Spring Data поддерживает многие разновидности баз данных NoSQL, включая MongoDB, Cassandra, Couchbase, Neo4j, Redis и др. И к счастью, для работы с ними используются практически одинаковые модели программирования.

В одной главе невозможно охватить все базы данных, которые поддерживает Spring Data. Но чтобы дать вам почувствовать «вкус разнообразия» Spring Data, мы рассмотрим две популярные базы данных NoSQL, Cassandra и MongoDB, и посмотрим, как создавать репозитории в них. Начнем с создания репозитория в Cassandra с помощью Spring Data.

4.1 Репозитории в Cassandra

Cassandra – это распределенная, высокопроизводительная, постоянно доступная, в конечном итоге согласованная база данных NoSQL с секционированным колоночным хранилищем.

Весьма красочный набор прилагательных, описывающих базу данных, но каждое из них точно определяет возможности Cassandra. Если говорить простым языком, то Cassandra хранит записи в таблицах, которые распределены между узлами, находящимися в отношениях «один ко многим». Ни один узел не хранит все данные, но любую конкретную запись можно получить из нескольких разных узлов, что устраняет любую единую точку отказа.

Spring Data Cassandra предлагает автоматизированную поддержку репозиториев на основе базы данных Cassandra, в общих чертах очень напоминающую то, что предлагает Spring Data JPA для реляционных баз данных, но имеющую существенные отличия в деталях. Кроме того, Spring Data Cassandra предлагает аннотации для отображения типов данных предметной области в структуры базы данных.

Прежде чем приступить к изучению Cassandra, вы должны понять, что Cassandra, хотя и разделяет многие идеи реляционных баз данных, таких как Oracle и SQL Server, в действительности не является реляционной базой данных и во многом отличается от них. Далее я раскрою особенности работы с Cassandra с использованием Spring Data. Но рекомендую вам прочитать документацию с описанием этой базы данных (<http://cassandra.apache.org/doc/latest/>), чтобы получить полное представление о ее особенностях.

Начнем с включения Spring Data Cassandra в проект Taco Cloud.

4.1.1 Включение Spring Data Cassandra

Чтобы получить возможность использовать Spring Data Cassandra, нужно добавить начальную зависимость Spring Boot для нереактивной версии Spring Data Cassandra. На самом деле есть две отдельные начальные зависимости Spring Data Cassandra: одна с поддержкой реактивного программирования и другая – стандартная.

С приемами создания реактивных репозиториев мы познакомимся позже, в главе 15, а пока будем использовать нереактивную версию:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-cassandra</artifactId>
</dependency>
```

Эта зависимость также доступна в веб-приложении Initializr в виде флагка **Cassandra**.

Важно понимать, что эта зависимость замещает начальные зависимости Spring Data JPA и Spring Data JDBC, которые мы использовали в предыдущей главе. В этой главе вместо JPA или JDBC мы будем использовать Spring Data для сохранения данных Taco Cloud в базе данных Cassandra. Поэтому не забудьте удалить из спецификации сборки начальные зависимости Spring Data JPA или Spring Data JDBC и любые другие зависимости поддержки реляционных баз данных (например, драйверы JDBC или зависимость H2).

Начальная зависимость Spring Data Cassandra добавляет в проект несколько зависимостей, в частности библиотеку Spring Data Cassandra. Так как Spring Data Cassandra оказывается в пути поиска классов среды выполнения, механизм автоконфигурации запускает процедуру создания репозиториев Cassandra. Это означает, что мы можем начинать писать репозитории Cassandra с минимумом явных настроек.

Cassandra охватывает кластер узлов, которые все вместе действуют как единая система баз данных. Если у вас еще нет кластера Cassandra для работы, то для нужд разработки вы можете запустить кластер с одним узлом, используя Docker:

```
$ docker network create cassandra-net
$ docker run --name my-cassandra \
    --network cassandra-net \
    -p 9042:9042 \
    -d cassandra:latest
```

Эта команда запустит кластер с одним узлом и откроет порт узла (9042) на хост-компьютере, через который наше приложение сможет получить к нему доступ.

Но нам нужно добавить некоторые настройки, в частности определить имя *пространства ключей*, в котором будут работать репозитории. Однако прежде нужно создать такое пространство ключей.

ПРИМЕЧАНИЕ Пространство ключей в Cassandra обеспечивает группировку таблиц. Примерно так же группируются таблицы, представления и ограничения в реляционных базах данных.

Можно, конечно, настроить Spring Data Cassandra для автоматического создания пространства ключей, но обычно гораздо проще создать его вручную (или использовать существующее пространство ключей). Давайте создадим пространство ключей для приложения Taco Cloud, используя командную оболочку Cassandra CQL (Cassandra

Query Language – язык запросов Casandra). Вот как можно запустить оболочку CQL с помощью Docker:

```
$ docker run -it --network cassandra-net --rm cassandra cqlsh my-cassandra
```

ПРИМЕЧАНИЕ Если эта команда потерпит неудачу с сообщением об ошибке «Unable to connect to any servers» (Невозможно подключиться ни к одному серверу), подождите минуту или две и повторите попытку. Кластер Cassandra должен закончить процедуру запуска, прежде чем оболочка CQL сможет подключиться к нему.

Когда оболочка будет готова, введите команду `create keyspace`:

```
cqlsh> create keyspace tacocloud
... with replication={'class':'SimpleStrategy', 'replication_factor':1}
... and durable_writes=true;
```

Эта команда создаст пространство ключей с именем `tacocloud` с простой репликацией и надежной записью. Установив коэффициент репликации равным 1, мы просим Cassandra сохранять по одной копии каждой записи. Стратегия репликации определяет порядок создания копий записей. Стратегия `SimpleStrategy` хорошо подходит для случаев, когда все узлы кластера с базой данных Cassandra находятся в одном вычислительном центре (и для демонстрационного кода, конечно). Если узлы кластера разбросаны по нескольким вычислительным центрам, то можно использовать стратегию `NetworkTopologyStrategy`. За более подробной информацией об особенностях разных стратегий репликации и альтернативных способах создания пространств ключей обращайтесь к документации с описанием Cassandra.

Теперь, создав пространство ключей, мы должны настроить свойство `spring.data.cassandra.keyspace-name`, чтобы указать библиотеке Spring Data Cassandra, что она должна использовать это пространство ключей:

```
spring:
  data:
    cassandra:
      keyspace-name: taco_cloud
      schema-action: recreate
      local-datacenter: datacenter1
```

Здесь мы также присвоили свойству `spring.data.cassandra.schema-action` значение `recreate`. Это свойство особенно полезно при разработке, потому что гарантирует удаление и воссоздание любых таблиц и пользовательских типов при каждом запуске приложения. Значение по умолчанию, `none`, предписывает не предпринимать никаких действий со схемой базы данных и используется в промышленных

окружениях, когда не требуется удалять все таблицы при запуске приложения.

Наконец, свойство `spring.data.cassandra.local-datacenter` определяет имя локального вычислительного центра для настройки политики балансировки нагрузки Cassandra. В конфигурациях с одним узлом используется значение "datacenter1". За дополнительной информацией о политиках балансировки нагрузки и как настроить локальный вычислительный центр обращайтесь к справочной документации с описанием драйвера DataStax Cassandra (<http://mng.bz/XrQM>).

Это единственные свойства, которые понадобятся нам для работы с локально работающей базой данных Cassandra. Однако, в зависимости от конфигурации кластера Cassandra, вы можете настроить другие свойства.

По умолчанию Spring Data Cassandra предполагает, что база данных Cassandra работает локально и прослушивает порт 9042. Если это не так, то можно установить свойства `spring.data.cassandra.contact-points` и `spring.data.cassandra.port`, например так:

```
spring:
  data:
    cassandra:
      keyspace-name: tacocloud
      local-datacenter: datacenter1
      contact-points:
        - casshost-1.tacocloud.com
        - casshost-2.tacocloud.com
        - casshost-3.tacocloud.com
      port: 9043
```

Обратите внимание, что свойство `spring.data.cassandra.contact-points` должно определять имя (имена) хоста Cassandra. Точка контакта – это хост, на котором работает узел с Cassandra. По умолчанию этому свойству присваивается значение `localhost`, но вы можете присвоить ему список имен хостов. В таком случае Spring Data Cassandra будет соединяться с каждой точкой контакта, пока не найдет ту, к которой сможет подключиться. Это делается для того, чтобы в кластере Cassandra не было единой точки отказа и чтобы приложение могло подключиться к кластеру через одну из заданных точек контакта.

Вам также может потребоваться указать имя пользователя и пароль для вашего кластера Cassandra. Это можно сделать с помощью свойств `spring.data.cassandra.username` и `spring.data.cassandra.password`, как показано ниже:

```
spring:
  data:
    cassandra:
      ...
      username: tacocloud
      password: s3cr3tP455w0rd
```

Это единственные свойства, которые понадобятся для работы с локальной базой данных Cassandra. Однако в зависимости от конфигурации кластера Cassandra вам также может понадобиться установить другие свойства.

Теперь, после включения и настройки Spring Data Cassandra, мы почти готовы отобразить типы данных предметной области в таблицы Cassandra и реализовать репозиторий. Но сначала отступим на шаг назад и рассмотрим некоторые особенности моделирования данных в Cassandra.

4.1.2 Моделирование данных в Cassandra

Как я уже упоминал, Cassandra сильно отличается от реляционных баз данных. Прежде чем начать отображать типы данных предметной области в таблицы Cassandra, важно понять, чем моделирование данных в Cassandra отличается от моделирования в реляционных базах данных.

Вот некоторые важные особенности моделирования данных в Cassandra, которые нужно знать:

- таблицы Cassandra могут иметь любое количество столбцов, но от записей не требуется использовать их все;
- базы данных Cassandra разделены на несколько сегментов, или разделов. Любая запись в данной таблице может храниться в одном или нескольких разделах, но весьма маловероятно, что каждый из разделов будет хранить все записи;
- таблицы в Cassandra имеют два типа ключей: ключи сегментирования и ключи кластеризации. Хеш-операции выполняются с ключом сегментирования для каждой записи, чтобы определить, какой раздел(ы) будет хранить эту запись. Ключи кластеризации определяют порядок, в каком записи будут храниться в разделе (этот порядок может не совпадать с порядком, в каком записи могут возвращаться в результатах запроса). Более полную информацию о моделировании данных в Cassandra, разделах, кластерах и соответствующих ключах ищите в документации (<http://mng.bz/yJ6E>);
- Cassandra оптимизирована для операций чтения. По этой причине обычно желательно, чтобы таблицы были сильно денормализованы, а данные повторялись в нескольких таблицах. (Например, информация о клиентах может храниться в таблице клиентов и повторяться в таблице заказов, размещенных клиентами.)

Проще говоря, чтобы адаптировать типы данных предметной области Taco Cloud для работы с Cassandra, недостаточно заменить несколько аннотаций JPA аннотациями Cassandra. Нам придется переосмыслить саму модель данных.

4.1.3 Отображение типов данных предметной области для хранения в Cassandra

В главе 3 мы снабдили свои типы данных предметной области (`Taco`, `Ingredient`, `TacoOrder` и т. д.) аннотациями JPA. Эти аннотации отображали типы данных в объекты, которые можно сохранить в реляционной базе данных. Эти аннотации непригодны для работы с Cassandra, однако в Spring Data Cassandra имеется свой набор аннотаций, предназначенных для той же цели.

Начнем с класса `Ingredient`, потому что его проще всего адаптировать для хранения в Cassandra. Вот как выглядит новый класс `Ingredient`:

```
package tacos;

import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.mapping.Table;

import lombok.AccessLevel;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Table("ingredients")
public class Ingredient {

    @PrimaryKey
    private String id;
    private String name;
    private Type type;

    public enum Type {
        WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
    }

}
```

Класс `Ingredient`, похоже, противоречит всему, что я сказал о простой замене нескольких аннотаций. Вместо аннотации `@Entity`, которую мы добавили на уровне класса для поддержки JPA, используется аннотация `@Table`, сообщающая, что экземпляры `Ingredient` теперь должны храниться в таблице `ingredients`. А аннотацию `@Id` у свойства `id` заменила аннотация `@PrimaryKey`. Пока похоже, что мы лишь поменяли несколько аннотаций.

Но давайте не будем заблуждаться. Класс `Ingredient` – один из самых простых типов предметной области. Гораздо больше изменений нас поджидает в классе `Taco`, как показано в листинге 4.1.

Листинг 4.1 Аннотирование класса Taco для подготовки к хранению в Cassandra

```
package tacos;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.UUID;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.springframework.data.cassandra.core.cql.Ordering;
import org.springframework.data.cassandra.core.cql.PrimaryKeyType;
import org.springframework.data.cassandra.core.mapping.Column;
import org.springframework.data.cassandra.core.mapping.PrimaryKeyColumn;
import org.springframework.data.cassandra.core.mapping.Table;

import com.datastax.oss.driver.api.core.uuid.Uuids;

import lombok.Data;

@Data
@Table("tacos") ← Хранить в таблице "tacos"
public class Taco { ← Определение
    @PrimaryKeyColumn(type=PrimaryKeyType.PARTITIONED) ← ключа раздела
    private UUID id = Uuids.timeBased();

    @NotNull
    @Size(min = 5, message = "Name must be at least 5 characters long")
    private String name; ← Определение ключа
    @PrimaryKeyColumn(type=PrimaryKeyType.CLUSTERED, ← кластеризации
                      ordering=Ordering.DESCENDING)
    private Date createdAt = new Date();

    @Size(min=1, message="You must choose at least 1 ingredient") ← Отображает
    @Column("ingredients") ← список в столбец
    private List<IngredientUDT> ingredients = new ArrayList<>(); ← "ingredients"

    public void addIngredient(Ingredient ingredient) {
        this.ingredients.add(TacoUDRUtils.toIngredientUDT(ingredient));
    }

}
```

Как видите, отображение класса Taco выглядит немного сложнее. Здесь также используется аннотация `@Table`, определяющая имя таблицы `tacos`, в которой должны храниться экземпляры Taco. Но на этом сходство с классом `Ingredient` заканчивается.

Свойство `id` по-прежнему является первичным ключом, но это только один из двух столбцов, образующих первичный ключ. В частности, обратите внимание, что свойство `id` снабжено аннотацией

@PrimaryKeyColumn с атрибутом type=PrimaryKeyType.PARTITIONED. Он указывает, что свойство `id` играет роль ключа раздела, используемого для выбора раздела или разделов, где Cassandra будет хранить записи с экземплярами Taco.

Также отметьте, что свойство `id` теперь имеет тип UUID вместо Long. В этом нет особой необходимости, но обычно свойства, хранящие генерированное уникальное значение идентификатора, имеют тип UUID. Кроме того, значения типа UUID генерируются для новых объектов на основе текущего времени (но его можно изменить при чтении существующего объекта из базы данных).

Далее следует определение свойства `createdAt`, которое служит вторым столбцом первичного ключа. Но в этом случае атрибут `type` в аннотации @PrimaryKeyColumn имеет значение PrimaryKeyType.CLUSTERED, вследствие чего свойство `createdAt` становится ключом кластеризации. Как упоминалось выше, ключи кластеризации используются для определения порядка хранения записей в разделе – в данном случае в порядке убывания, поэтому в пределах данного раздела более новые записи будут помещаться в начало таблицы `tacos`.

Наконец, свойство `ingredients` теперь объявлено как список объектов типа IngredientUDT, а не Ingredient. Как вы, наверное, помните, таблицы в Cassandra сильно денормализованы и могут содержать повторяющиеся данные из других таблиц. Таблица `ingredient` будет хранить записи для всех доступных ингредиентов, но ингредиенты из списка `ingredients` в данном экземпляре Taco будут продублированы в столбце `ingredients`. Вместо ссылок на записи в таблице `ingredients` свойство `ingredients` будет хранить полные данные для каждого выбранного ингредиента.

Но зачем понадобилось вводить новый класс IngredientUDT? Почему нельзя просто использовать класс Ingredient? Дело в том, что столбцы, хранящие коллекции данных, такие как столбец `ingredients`, должны быть коллекциями встроенных типов (целых чисел, строк и т. д.) или типов, определяемых пользователем.

В Cassandra пользовательские типы позволяют объявлять столбцы таблицы с более богатыми типами. Часто они используются как денормализованный аналог реляционных внешних ключей. В отличие от внешних ключей, которые хранят только ссылки на записи в других таблицах, столбцы с пользовательскими типами хранят фактические данные, которые можно скопировать в другую таблицу. Соответственно, столбец `ingredients` в таблице `tacos` будет хранить коллекцию структур данных, определяющих сами ингредиенты.

Использовать класс Ingredient в качестве типа, определяемого пользователем, нельзя, потому что аннотация @Table уже отобразила его как хранимую сущность в Cassandra. Поэтому нам пришлось объявить новый класс, чтобы определить, как ингредиенты будут храниться в столбце `ingredients` таблицы `tacos`. IngredientUDT (где UDT означает User-Defined Type – определяемый пользователем тип) – это класс, созданный специально для данной цели:

```

package tacos;

import org.springframework.data.cassandra.core.mapping.UserDefinedType;

import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Data
@RequiredArgsConstructor
@NoArgsConstructor(access = AccessLevel.PRIVATE, force = true)
@UserDefinedType("ingredient")
public class IngredientUDT {

    private final String name;

    private final Ingredient.Type type;

}

```

Класс `IngredientUDT` очень похож на `Ingredient`, но требования к его отображению намного проще. Он снабжен аннотацией `@UserDefinedType`, идентифицирующей его как тип, определяемый пользователем. Но в остальном это простой класс с несколькими свойствами.

Обратите также внимание, что в классе `IngredientUDT` отсутствует свойство `id`. Вообще говоря, экземпляры `IngredientUDT` могут хранить копии свойства `id` из исходного экземпляра `Ingredient`, но в этом нет никакой необходимости. На самом деле пользовательский тип может включать любые свойства по вашему желанию, но от него не требуется обладать свойством, уникально идентифицирующим его экземпляры.

Я понимаю, что порой сложно представить, как данные пользовательского типа соотносятся с данными, хранящимися в таблице. Поэтому я решил привести схему всей базы данных для Taco Cloud, включая пользовательские типы (рис. 4.1).

Обратите внимание, что теперь экземпляр `Taco` имеет список данных только что созданного пользовательского типа, скопированных из объектов `Ingredient`. Вместе с `Taco` в таблице `tacos` сохраняется сам объект `Taco` и его список объектов `IngredientUDT`. То есть список объектов `IngredientUDT` целиком сохраняется в столбце `ingredients`.

Взгляд под другим углом – с точки зрения запроса записей из таблицы `tacos` – может помочь вам лучше понять, как используются определяемые пользователем типы. Выполнив CQL-запрос с помощью инструмента `cqlsh`, входящего в состав Cassandra, вы получите следующие результаты:

```

cqlsh:tacocloud> select id, name, createdat, ingredients from tacos;
id      | name      | createdat | ingredients
-----+-----+-----+-----
827390... | Carnivore | 2018-04...| [{name: 'Flour Tortilla', type: 'WRAP'},
```

```
{name: 'Carnitas', type: 'PROTEIN'},
{name: 'Sour Cream', type: 'SAUCE'},
{name: 'Salsa', type: 'SAUCE'},
{name: 'Cheddar', type: 'CHEESE']}  
(1 rows)
```

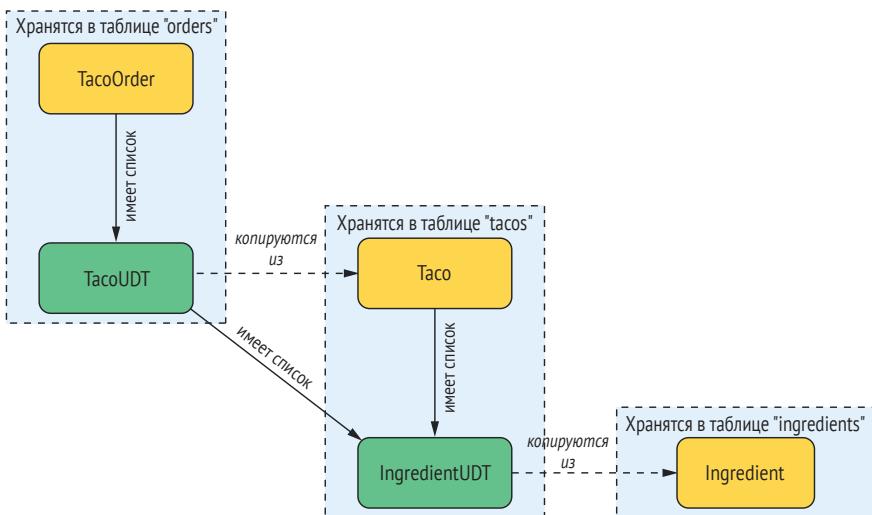


Рис. 4.1 Вместо внешних ключей и соединений в Cassandra используется прием денормализации таблиц, которые хранят копии данных пользовательских типов из связанных таблиц

Как видите, столбцы `id`, `name` и `createdat` содержат простые значения. В этом они мало отличаются от того, что можно ожидать от аналогичного запроса к реляционной базе данных. Но столбец `ingredients` стоит особняком. Поскольку он определен как хранящий коллекцию данных пользовательского типа `IngredientUDT`, его значение отображается как массив JSON, заполненный объектами JSON.

Вы наверняка заметили другие пользовательские типы на рис. 4.1, в частности используемые классом `TacoOrder`. Вам часто придется определять такие типы при использовании Cassandra. В листинге 4.2 показано определение класса `TacoOrder`, адаптированного для хранения в Cassandra.

Листинг 4.2 Класс `TacoOrder`, адаптированный для хранения в Cassandra

```
package tacos;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;
```

```

import java.util.UUID;

import javax.validation.constraints.Digits;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Pattern;

import org.hibernate.validator.constraints.CreditCardNumber;
import org.springframework.data.cassandra.core.mapping.Column;
import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.mapping.Table;

import com.datastax.oss.driver.api.core.uuid.Uuids;

import lombok.Data;

@Data
@Table("orders") ← Хранить в таблице "orders"
public class TacoOrder implements Serializable {

    private static final long serialVersionUID = 1L;

    @PrimaryKey ← Определение первичного ключа
    private UUID id = Uuids.timeBased();

    private Date placedAt = new Date();

    // свойства с адресом доставки и номером кредитной карты опущены для краткости

    @Column("tacos") ← Хранить список в столбце "tacos"
    private List<TacoUDT> tacos = new ArrayList<>();

    public void addTaco(TacoUDT taco) {
        this.tacos.add(taco);
    }

}

```

В листинге 4.2 намеренно опущены многие свойства TacoOrder, которые не имеют отношения к обсуждению особенностей моделирования данных в Cassandra. Я оставил лишь несколько свойств, заслуживающих особого внимания, подобно свойствам в классе Taco. Аннотация @Table отображает класс TacoOrder в таблицу orders. В данном случае нас не волнует порядок хранения данных, поэтому свойство id аннотировано лишь аннотацией @PrimaryKey, превращающей это свойство в ключ раздела и ключ кластеризации с порядком следования по умолчанию.

Свойство tacos интересно тем, что оно имеет тип List<TacoUDT>, а не List<Taco>. Связь между TacoOrder и Taco/TacoUDT здесь аналогична связи между Taco и Ingredient/IngredientUDT. То есть вместо объединения данных из записей строк в отдельной таблице с помощью внешних ключей таблица orders будет хранить полные списки с данными о тако, что обеспечит высокую скорость чтения.

Класс TacoUDT очень похож на класс IngredientUDT, но включает коллекцию, ссылающуюся на другой пользовательский тип, как показано ниже:

```
package tacos;

import java.util.List;
import org.springframework.data.cassandra.core.mapping.UserDefinedType;
import lombok.Data;

@Data
@UserDefinedType("taco")
public class TacoUDT {

    private final String name;
    private final List<IngredientUDT> ingredients;

}
```

Было бы неплохо иметь возможность повторно использовать те же классы предметной области, что были созданы в главе 3, или хотя бы заменить некоторые аннотации JPA аннотациями Cassandra, но природа Cassandra такова, что требует переосмыслиния моделирования данных.

А теперь, подготовив типы данных предметной области, мы готовы приступить к определению репозиториев.

4.1.4 Определение репозиториев Cassandra

Как мы видели в главе 3, чтобы определить репозиторий в Spring Data, достаточно объявить интерфейс, наследующий один из базовых интерфейсов репозиториев в Spring Data, и, может быть, добавить дополнительные методы запросов. Как оказывается, определение репозиториев для Cassandra мало чем отличается.

На самом деле уже написанные нами репозитории нужно лишь немного подправить, чтобы заставить их работать с Cassandra. Например, рассмотрим репозиторий IngredientRepository, созданный в главе 3:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Ingredient;

public interface IngredientRepository
    extends CrudRepository<Ingredient, String> {
}
```

Благодаря наследованию CrudRepository, как показано здесь, наш репозиторий IngredientRepository уже готов сохранять объекты

Ingredient, если они будут иметь свойство идентификатора (или, в случае с Cassandra, свойство первичного ключа) строкового типа. Идеально! Нам не нужно вносить никаких изменений в IngredientRepository.

Репозиторий OrderRepository нужно лишь немного изменить: заменить параметр типа Long для идентификатора, указанного при наследовании CrudRepository, на UUID:

```
package tacos.data;  
  
import java.util.UUID;  
  
import org.springframework.data.repository.CrudRepository;  
  
import tacos.TacoOrder;  
  
public interface OrderRepository  
    extends CrudRepository<TacoOrder, UUID> {  
}
```

Cassandra обладает широкими возможностями, и благодаря возможностям Spring Data их можно с успехом использовать в приложениях Spring.

А теперь переключим наше внимание на другую базу данных, для которой имеется поддержка репозиториев Spring Data: MongoDB.

4.2 Определение репозиториев MongoDB

MongoDB – еще одна известная база данных NoSQL. В отличие от колоночной базы данных Cassandra, MongoDB считается документной базой данных. В частности, MongoDB хранит документы в формате BSON (Binary JSON), которые можно запрашивать и извлекать подобно обычным данным в любой другой базе данных.

Так же как в случае с Cassandra, важно понимать, что MongoDB – нереляционная база данных. Управление кластером серверов MongoDB, а также моделирование данных требуют использования иных подходов, чем при работе с другими типами баз данных.

И все же благодаря Spring Data работа с MongoDB не сильно отличается от работы с JPA или Cassandra. Вы точно так же должны снабдить классы данных предметной области аннотациями, отображающими типы данных предметной области в структуры документов, и написать интерфейсы репозиториев, следуя практически той же модели программирования, которую мы видели на примерах репозиториев для JPA и Cassandra. Но, прежде чем что-то делать, нужно подключить Spring Data MongoDB к проекту приложения.

4.2.1 Включение Spring Data MongoDB

Чтобы получить возможность использовать Spring Data MongoDB, нужно добавить начальную зависимость Spring Data MongoDB в спецификацию сборки проекта. Подобно Spring Data Cassandra, поддержка Spring Data MongoDB предлагает на выбор две отдельные версии библиотеки: реактивную и нереактивную. С реактивной версией мы познакомимся в главе 13. А пока будем использовать нереактивную версию MongoDB:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>
        spring-boot-starter-data-mongodb
    </artifactId>
</dependency>
```

Эта зависимость также доступна в веб-приложении Spring Initializr в виде флагка **MongoDB** в разделе **NoSQL**.

После добавления начальной зависимости в сборку механизм автоконфигурации включит поддержку автоматических интерфейсов репозиториев в Spring Data, подобных тем, которые мы писали для JPA в главе 3 или для Cassandra выше в этой главе.

По умолчанию Spring Data MongoDB предполагает, что у вас есть сервер MongoDB, работающий локально и прослушивающий порт 27017. Если на вашем компьютере установлен Docker, то запустить сервер MongoDB можно простой командой:

```
$ docker run -p 27017:27017 -d mongo:latest
```

Но на этапе тестирования и разработки проще и удобнее использовать встраиваемую базу данных MongoDB. Для этого подключите Flapdoodle – встраиваемую версию MongoDB, – добавив следующую зависимость в спецификацию сборки:

```
<dependency>
    <groupId>de.flapdoodle.embed</groupId>
    <artifactId>de.flapdoodle.embed.mongo</artifactId>
    <!-- <scope>test</scope> -->
</dependency>
```

Встраиваемая база данных Flapdoodle обеспечивает такое же удобство работы с базой данных Mongo в памяти, как и H2 в отношении реляционных баз данных. То есть вам не придется запускать отдельный сервер баз данных, но все данные исчезнут после остановки приложения.

Встраиваемые базы данных удобны для разработки и тестирования, но при развертывании приложения в промышленном окружении вам придется изменить несколько свойств, чтобы библиотека

Spring Data MongoDB знала, где находится ваша база данных и как получить к ней доступ:

```
spring:
  data:
    mongodb:
      host: mongodb.tacocloud.com
      port: 27017
      username: tacocloud
      password: s3cr3tp455w0rd
      database: tacoclouddb
```

Не все эти свойства являются обязательными, но они позволяют настроить Spring Data MongoDB в случаях, если ваша база данных Mongo действует не локально. Вот какие роли играют эти свойства:

- *spring.data.mongodb.host* – имя хоста, где действует сервер Mongo (по умолчанию: `localhost`);
- *spring.data.mongodb.port* – номер порта, который прослушивается сервером Mongo (по умолчанию: 27017);
- *spring.data.mongodb.username* – имя пользователя для доступа к базе данных Mongo;
- *spring.data.mongodb.password* – пароль для доступа к базе данных Mongo;
- *spring.data.mongodb.database* – имя базы данных (по умолчанию: `test`).

Теперь, после включения и настройки Spring Data MongoDB, можно аннотировать типы данных предметной области, чтобы подготовить их к сохранению в виде документов в MongoDB.

4.2.2 Отображение типов данных предметной области в документы

Spring Data MongoDB предлагает свой набор аннотаций для отображения типов данных предметной области в документы MongoDB. Однако из всего набора таких аннотаций наиболее часто используются только четыре:

- `@Id` – объявляет свойство уникальным идентификатором документа (из Spring Data Commons);
- `@Document` – объявляет тип данных документом, который может храниться в MongoDB;
- `@Field` – определяет имя поля (и дополнительно порядок сортировки) для сохранения свойства в хранимом документе;
- `@Transient` – определяет свойство, которое не будет сохраняться в базе данных.

Из этих аннотаций только `@Id` и `@Document` абсолютно необходимы. Если свойства не снабдить аннотацией `@Field` или `@Transient`, то они будут сохраняться в полях с именами, повторяющими имена свойств.

Применив эти аннотации к классу `Ingredient`, получим следующее:

```
package tacos;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.AccessLevel;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@Document
@AllArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
public class Ingredient {

    @Id
    private String id;
    private String name;
    private Type type;

    public enum Type {
        WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
    }
}
```

Как видите, аннотация `@Document` помещается на уровне класса `Ingredient` и объявляет его документом, который можно хранить в базе данных Mongo. По умолчанию коллекции (аналог таблицы в реляционных базах данных) присваивается имя, соответствующее имени класса с первой буквой в нижнем регистре. Поскольку мы не указали иное имя, объекты `Ingredient` будут сохраняться в коллекции с именем `ingredient`. Но вы можете изменить это имя, добавив в аннотацию `@Document` атрибут `collection`:

```
@Data
@AllArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Document(collection="ingredients")
public class Ingredient {
    ...
}
```

Обратите также внимание, что свойство `id` снабжено аннотацией `@Id`. Эта аннотация объявляет свойство уникальным идентификатором хранимого документа. Аннотацию `@Id` можно применить к любому свойству с типом `Serializable`, включая `String` и `Long`. В данном случае в роли естественного идентификатора мы используем строковое свойство `id`, поэтому нет необходимости менять его тип.

Пока все хорошо. Но, как отмечалось выше в этой главе, `Ingredient` – это простой тип данных. Другие типы, такие как `Taco`, несколько

сложнее. Давайте посмотрим, как отобразить класс Taco и какие сюрпризы он может преподнести.

Подход к хранимым документам в MongoDB хорошо укладывается в идею предметно-ориентированного проектирования с сохранением корневых агрегатов. Документы в MongoDB, как правило, определяются как корни агрегатов, а составляющие агрегата – как вложенные документы.

Для Taco Cloud это означает следующее: так как объекты Taco всегда будут сохраняться только как составляющие агрегата TacoOrder, класс Taco не нужно снабжать аннотациями `@Document` и `@Id`. Он может оставаться свободным от любых аннотаций механизма хранения:

```
package tacos;

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

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import lombok.Data;

@Data
public class Taco {

    @NotNull
    @Size(min=5, message="Name must be at least 5 characters long")
    private String name;

    private Date createdAt = new Date();

    @Size(min=1, message="You must choose at least 1 ingredient")
    private List<Ingredient> ingredients = new ArrayList<>();

    public void addIngredient(Ingredient ingredient) {
        this.ingredients.add(ingredient);
    }

}
```

Однако класс TacoOrder – это корень агрегата и должен снабжаться аннотацией `@Document`, а также иметь свойство с аннотацией `@Id`:

```
package tacos;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import javax.validation.constraints.Digits;
import javax.validation.constraints.NotBlank;
```

```
import javax.validation.constraints.Pattern;

import org.hibernate.validator.constraints.CreditCardNumber;
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Data;

@Data
@Document
public class TacoOrder implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private String id;

    private Date placedAt = new Date();

    // другие свойства опущены для краткости

    private List<Taco> tacos = new ArrayList<>();

    public void addTaco(Taco taco) {
        this.tacos.add(taco);
    }

}
```

Для краткости опустил некоторые поля с адресом доставки и информацией о кредитной карте. Но из того, что осталось, ясно видно, что нам нужны только аннотации `@Document` и `@Id`, как и в случае с другими типами данных предметной области.

Однако обратите внимание, что тип свойства `id` изменился на `String` (в отличие от `Long` в версии JPA или `UUID` в версии Cassandra). Как я уже говорил, аннотацию `@Id` можно применить к любому типу `Serializable`. Но, выбрав тип `String` для свойства-идентификатора, мы получаем дополнительную выгоду – база данных Mongo автоматически будет присваивать ему значение при сохранении (при условии что оно имеет значение `null`). Выбирая тип `String`, мы получаем значение идентификатора от базы данных, и отпадает необходимость заботиться о выборе значения для этого свойства вручную.

Конечно, есть несколько сложных и необычных случаев, однако чаще для отображения предметных типов для хранения в MongoDB достаточно аннотаций `@Document` и `@Id`, иногда `@Field` или `@Transient`. И именно к таким частым случаям относится Taco Cloud.

Теперь осталось только определить интерфейсы репозитория.

4.2.3 Определение репозиториев MongoDB

Spring Data MongoDB предлагает автоматическую поддержку репозиториев, так же как Spring Data JPA и Spring Data Cassandra.

Начнем с определения репозитория для хранения объектов Ingredient в виде документов. Как и прежде, можно объявить, что репозиторий IngredientRepository наследует CrudRepository:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.Ingredient;

public interface IngredientRepository
    extends CrudRepository<Ingredient, String> {
}
```

Но подождите! Это объявление выглядит *идентично* объявлению интерфейса IngredientRepository в разделе 4.1, который мы писали для Cassandra! И действительно, это тот же интерфейс, без каких-либо изменений. Это подчеркивает одно из преимуществ наследования CrudRepository – данный интерфейс обеспечивает переносимость между разными типами баз данных и одинаково хорошо подходит и для MongoDB, и для Cassandra.

Реализация интерфейса OrderRepository тоже довольно проста:

```
package tacos.data;

import org.springframework.data.repository.CrudRepository;

import tacos.TacoOrder;

public interface OrderRepository
    extends CrudRepository<TacoOrder, String> {
}
```

Репозиторий OrderRepository тоже наследует интерфейс CrudRepository, чтобы получить оптимизированные версии методов insert(). В остальном этот репозиторий ничем особенным не отличается от некоторых других репозиториев, которые мы уже определили. Но обратите внимание, что при наследовании CrudRepository параметр типа для идентификатора теперь имеет значение String, а не Long (как в JPA) или UUID (как для Cassandra). Это отражает изменение, которое мы внесли в TacoOrder для поддержки автоматического выбора значений идентификаторов.

В общем и целом использование Spring Data MongoDB не сильно отличается от использования других проектов Spring Data, с которыми мы работали. Да, типы данных предметной области аннотируются по-разному, но при наследовании CrudRepository интерфейсы репозиториев получаются практически идентичными, кроме параметра типа для идентификатора.

Итоги

- Spring Data поддерживает репозитории для разных баз данных NoSQL, включая Cassandra, MongoDB, Neo4j и Redis.
- Модель программирования для создания репозиториев мало отличается для разных баз данных.
- Работа с нереляционными базами данных требует понимания особенностей моделирования данных и их хранения в базе данных.

Безопасность в *Spring*



В этой главе рассматриваются следующие темы:

- автоконфигурация Spring Security;
- определение хранилища ученых записей;
- настройка страницы входа;
- защита от атак CSRF;
- знай своего пользователя.

Вы когда-нибудь замечали, что большинство людей в телевизионных комедиях не запирают свои двери? Во времена комедийного сериала «Leave It to Beaver»¹ люди нередко оставляли свои двери незапертными. Но в наше время, когда мы особенно заботимся о конфиденциальности и безопасности, кажется безумием, что телевизионные персонажи никак не пытаются препятствовать доступу посторонних в свои квартиры и дома.

Информация – это, пожалуй, самое ценное из того, что у нас сейчас есть; мошенники ищут способы украсть наши данные, проникая в незащищенные приложения. Как разработчики программного обеспечения мы должны предпринимать все меры для защиты информации

¹ Комедийный сериал, демонстрировавшийся в США в 1957–1963 годах. – *Прим. перев.*

в наших приложениях. Будь то учетные записи электронной почты, защищенные именами пользователей и паролями, или брокерская учетная запись, защищенная PIN-кодом, безопасность является важнейшим аспектом большинства приложений.

5.1 Включение Spring Security

Самый первый шаг добавления поддержки безопасности в приложение Spring – включение начальной зависимости `spring-boot-starter-security` безопасности Spring Boot в спецификацию сборки. Для этого добавьте в файл `pom.xml` следующий элемент `<dependency>`:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Если вы используете Spring Tool Suite, то можно поступить проще. Щелкните правой кнопкой мыши на файле `pom.xml` и выберите в контекстном меню **Edit Starters** (Редактировать начальные зависимости). В открывшемся диалоге установите флажок **Spring Security** в категории **Security** (Безопасность), как показано на рис. 5.1.

Хотите верьте, хотите нет, но эта зависимость – единственное, что нужно для защиты приложения. В момент запуска приложения механизм автоконфигурации обнаружит присутствие Spring Security в пути поиска классов и настроит базовую конфигурацию безопасности.

Для эксперимента запустите приложение и попробуйте открыть домашнюю страницу (или любую другую страницу, если на то пошло). Перед вами появится простая страница входа, которая выглядит примерно так, как показано на рис. 5.2, предлагающая пройти аутентификацию.

СОВЕТ Работа в режиме «инкогнито». Возможно, вам будет полезно установить в браузере приватный режим или режим «инкогнито» при тестировании безопасности вручную. Это гарантирует создание нового сеанса каждый раз, когда открывается приватное. При этом вам придется каждый раз входить в приложение, но будьте уверены, что любые изменения, внесенные в систему безопасности, будут применены и не останется ничего от старого сеанса, мешающего увидеть изменения.

Для аутентификации нужно ввести имя пользователя и пароль. Имя пользователя – `user`, а ноль генерируется случайным образом и записывается в файл журнала приложения. Запись в журнале будет выглядеть примерно так:

```
Using generated security password: 087cfcc6a-027d-44bc-95d7-cbb3a798a1ea
```

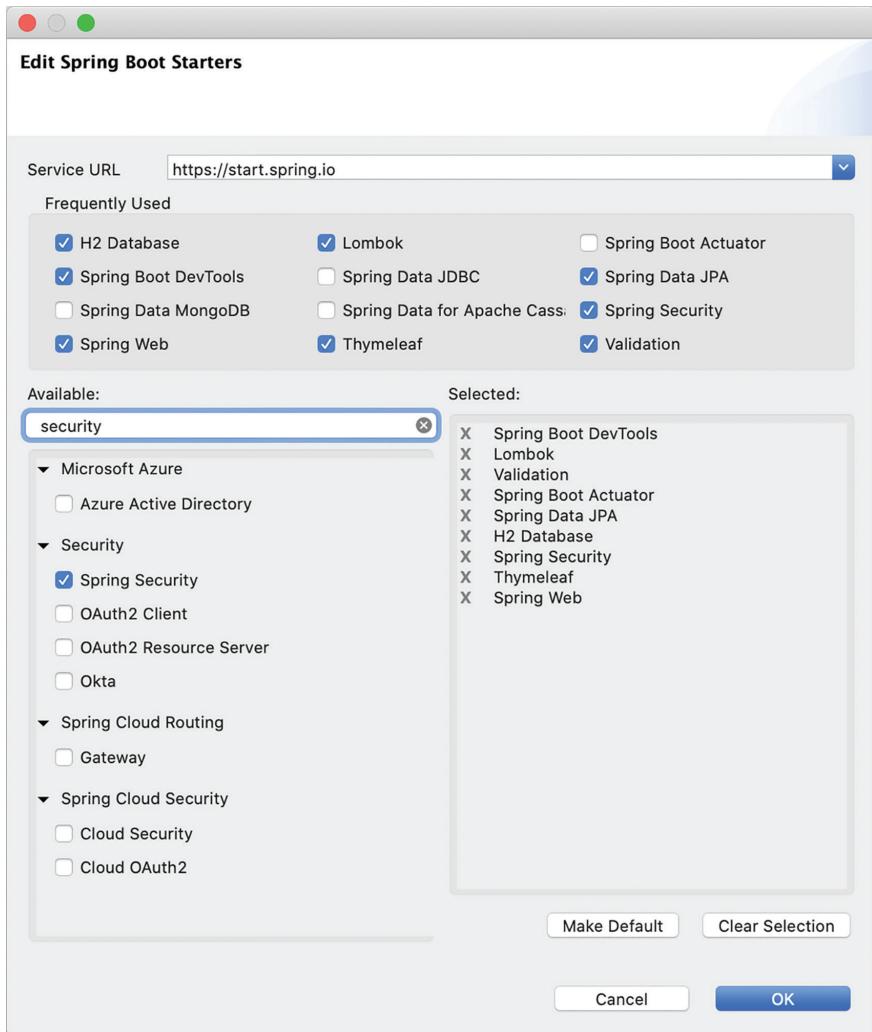


Рис. 5.1 Добавление начальной зависимости поддержки безопасности в Spring Tool Suite

После ввода правильных имени пользователя и пароля вы получите доступ к приложению.

Кажется, что обеспечить безопасность приложений Spring довольно просто и теперь, когда приложение Taco Cloud защищено, я мог бы закончить эту главу и перейти к следующей теме. Но не будем торопиться и посмотрим, какую защиту обеспечивает механизм автоконфигурации.

Не предпринимая ничего, кроме добавления начальной зависимости в спецификацию сборки проекта, вы получаете следующее:

- все пути HTTP-запросов требуют аутентификации;
- никаких особых ролей или полномочий не предусматривается;

- для аутентификации предлагается простая страница входа;
- есть только один пользователь – *user*.

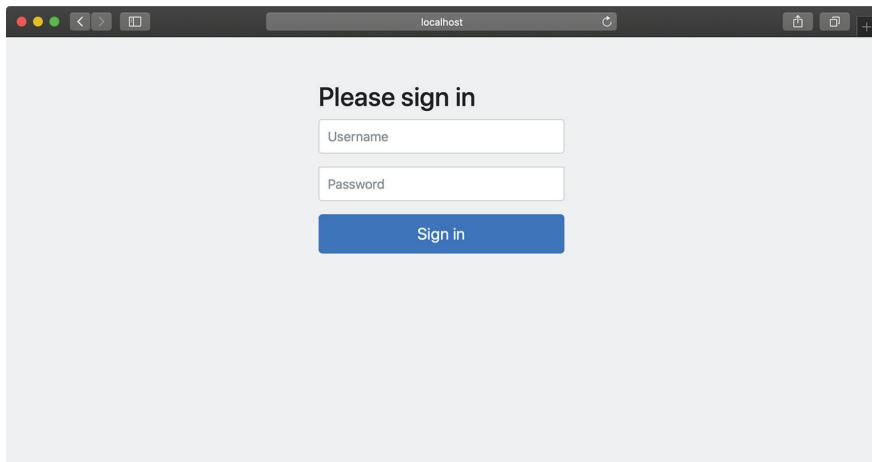


Рис. 5.2 По умолчанию Spring Security отображает простую страницу входа

Неплохо для начала, но, как мне кажется, потребности большинства приложений (включая Taco Cloud) в сфере безопасности будут сильно отличаться от этой элементарной поддержки.

Нам многое еще нужно сделать, чтобы должным образом защитить приложение Taco Cloud. Как минимум мы должны настроить Spring Security, чтобы:

- сформировать страницу входа, соответствующую веб-сайту;
- добавить поддержку множества пользователей и реализовать страницу регистрации, чтобы новые клиенты Taco Cloud могли зарегистрироваться;
- применить разные правила безопасности для разных путей в запросах. Домашняя страница и страница регистрации, например, вообще не должны требовать аутентификации.

Чтобы удовлетворить перечисленные потребности, нам придется явно определить некоторую конфигурацию, переопределяющую настройки, созданные механизмом автоконфигурации. Начнем с настройки правильного хранилища учетных записей пользователей, чтобы приложение могло обслуживать множество пользователей.

5.2 Настройка аутентификации

На протяжении многих лет существовало несколько способов настройки Spring Security, включая длинные конфигурационные XML-файлы. К счастью, в последних версиях Spring Security появилась возможность описывать настройки на языке Java.

Прежде чем эта глава закончится, вы научитесь настраивать безопасность Taco Cloud на Java. Но для начала давайте упростим себе задачу, написав класс конфигурации, показанный в листинге 5.1.

Листинг 5.1 Базовый класс конфигурации для Spring Security

```
package tacos.security;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
public class SecurityConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

}
```

Что вам дает этот базовый класс конфигурации безопасности? На самом деле не так много. Но самое главное – он объявляет bean-компонент `PasswordEncoder`, который мы будем использовать при создании новых пользователей и при аутентификации. В данном случае класс использует `BCryptPasswordEncoder`, один из нескольких средств шифрования паролей, имеющихся в составе Spring Security, включая следующие:

- `BCryptPasswordEncoder` – применяет надежное шифрование bcrypt;
- `NoOpPasswordEncoder` – не применяет шифрования;
- `Pbkdf2PasswordEncoder` – применяет шифрование PBKDF2;
- `SCryptPasswordEncoder` – применяет шифрование Scrypt;
- `StandardPasswordEncoder` – применяет шифрование SHA-256.

Независимо от выбора инструмента шифрования паролей важно понимать, что пароли хранятся в базе данных в зашифрованном виде и их нельзя дешифровать. Пароль, который пользователь вводит при входе в систему, шифруется с использованием того же алгоритма, а затем сравнивается с шифрованным паролем в базе данных. Сравнение выполняет метод `matches()` объекта `PasswordEncoder`.

Какой инструмент шифрования паролей выбрать?

Не все инструменты шифрования паролей одинаковы. Вы должны составить каждый алгоритм шифрования с вашими целями в области безопасности и выбрать наиболее подходящий для вас. Но есть такие инструменты шифрования, которые не следует использовать в промышленных приложениях.

NoOpPasswordEncoder вообще не применяет шифрование. Поэтому он не подходит для использования в производстве, но его можно применять для тестирования. StandardPasswordEncoder считается недостаточно безопасным и фактически устарел.

Вместо этих двух инструментов лучше выбрать какой-нибудь другой, более безопасный. В примерах в этой книге мы будем использовать BCryptPasswordEncoder.

Кроме инструмента шифрования паролей, мы добавим в этот конфигурационный класс дополнительные bean-компоненты, соответствующие конкретным требованиям к безопасности нашего приложения. Начнем с настройки хранилища учетных записей пользователей.

Чтобы настроить хранилище учетных записей пользователей для их аутентификации, нам понадобится bean-компонент UserDetailsService. Интерфейс UserDetailsService относительно прост и имеет всего один метод, который нужно реализовать. Вот как выглядит UserDetailsService:

```
public interface UserDetailsService {  
  
    UserDetails loadUserByUsername(String username) throws  
        UsernameNotFoundException;  
  
}
```

Метод loadUserByUsername() принимает имя пользователя и отыскивает соответствующий объект UserDetails. Если учетная запись с таким именем пользователя не будет найдена, то метод генерирует исключение UsernameNotFoundException.

Как оказывается, в Spring Security есть несколько готовых реализаций UserDetailsService, в том числе:

- хранилище учетных записей в памяти;
- хранилище учетных записей JDBC;
- хранилище учетных записей LDAP.

Но вы, если понадобится, можете создать свою реализацию, отвечающую вашим конкретным потребностям.

Для начала попробуем реализацию UserDetailsService в памяти.

5.2.1 Служба хранения сведений о пользователях в памяти

Единственное место, где эта служба может хранить информацию об учетных записях пользователей, – это память. Предположим, у нас всего несколько учетных записей, которые вряд ли изменятся в будущем. В этом случае было бы более чем достаточно определить эти учетные записи как часть конфигурации безопасности.

Следующий метод bean-компоненты (листинг 5.2) показывает, как для этой цели создать `InMemoryUserDetailsManager` с двумя пользователями «buzz» и «woody».

Листинг 5.2 Определение учетных записей для службы учетных записей в памяти

```
@Bean
public UserDetailsService userDetailsService(PasswordEncoder encoder) {
    List<UserDetails> usersList = new ArrayList<>();
    usersList.add(new User(
        "buzz", encoder.encode("password"),
        Arrays.asList(new SimpleGrantedAuthority("ROLE_USER"))));
    usersList.add(new User(
        "woody", encoder.encode("password"),
        Arrays.asList(new SimpleGrantedAuthority("ROLE_USER"))));
    return new InMemoryUserDetailsManager(usersList);
}
```

Здесь создается список объектов `User`, каждый из которых имеет свойства для хранения имени пользователя, пароля и списка привилегий. Далее на основе этого списка создается `InMemoryUserDetailsManager`.

Если попробовать сейчас запустить приложение, то вы сможете войти в систему как «woody» или «buzz», используя пароль `password`.

Служба хранения учетных записей пользователей в памяти удобна для тестирования или для очень простых приложений, но не предусматривает простой возможности редактирования учетных записей. Если вам понадобится добавить, удалить или изменить привилегии пользователя, то придется внести необходимые изменения, а затем вновь собрать и развернуть приложение.

Нам в приложении Taco Cloud нужно, чтобы клиенты имели возможность регистрироваться и управлять своими учетными записями. Но служба хранения учетных записей в памяти не позволяет этого, поэтому сделаем еще шаг и посмотрим, как создать свою реализацию `UserDetailsService` с поддержкой базы данных в качестве хранилища учетных записей.

5.2.2 Настройка аутентификации пользователя

В предыдущей главе мы остановились на использовании Spring Data JPA в качестве механизма хранения всех данных о тако, ингредиентах и заказах. Поэтому есть смысл использовать тот же механизм и для хранения учетных записей пользователей. В этом случае данные будут находиться в реляционной базе данных, и мы сможем использовать аутентификацию JDBC. Но было бы еще лучше использовать для этой цели репозиторий Spring Data JPA.

Однако давайте по порядку. Сначала создадим тип данных предметной области и интерфейс репозитория, представляющий и сохраняющий информацию о пользователях.

ОПРЕДЕЛЕНИЕ ТИПА ДАННЫХ И РЕПОЗИТОРИЯ

При регистрации в приложении Taco Cloud клиенты должны сообщить о себе не только имя пользователя и пароль. Они также должны указать свое полное имя, адрес и номер телефона. Эта информация может использоваться для различных целей, в том числе для предварительного заполнения формы заказа (не говоря уже о потенциальных маркетинговых возможностях).

Чтобы собрать всю эту информацию, определим класс `User`, как показано в листинге 5.3.

Листинг 5.3 Определение класса учетной записи пользователя

```
package tacos;

import java.util.Arrays;
import java.util.Collection;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import lombok.AccessLevel;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

@Entity
@Data
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@RequiredArgsConstructor
public class User implements UserDetails {

    private static final long serialVersionUID = 1L;

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    private Long id;

    private final String username;
    private final String password;
    private final String fullname;
    private final String street;
    private final String city;
    private final String state;
    private final String zip;
    private final String phoneNumber;

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
```

```

        return Arrays.asList(new SimpleGrantedAuthority("ROLE_USER"));
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}

```

Первое, что следует отметить: этот тип User не совпадает с классом User, который мы использовали при создании службы хранения учетных записей в памяти. Новый тип содержит больше подробностей о пользователе, которые понадобятся для выполнения заказов, включая адрес и контактную информацию.

Возможно, вы обратили внимание, что этот класс User немного сложнее других объектов, созданных в главе 3. Кроме определения нескольких свойств, класс User также объявляет, что реализует интерфейс UserDetails из Spring Security.

Реализации UserDetails предоставляют фреймворку безопасности некоторую важную информацию о пользователе, например какие привилегии предоставлены пользователю и активна ли учетная запись.

Метод getAuthorities() должен возвращать набор привилегий пользователя. Различные методы is* возвращают логическое значение, сообщая состояние учетной записи пользователя: активна, заблокирована или срок ее действия истек.

Для нашего объекта User метод getAuthorities() просто возвращает коллекцию, сообщающую, что все пользователи имеют привилегию ROLE_USER. Кроме того, в приложении Taco Cloud, по крайней мере пока, не предусматривается отключение пользователей, поэтому все методы is* возвращают true, сообщая, что пользователи активны.

Теперь, определив тип User, можно перейти к интерфейсу репозитория:

```

package tacos.data;

import org.springframework.data.repository.CrudRepository;

```

```
import tacos.User;

public interface UserRepository extends CrudRepository<User, Long> {
    User findByUsername(String username);
}
```

В дополнение к операциям создания/чтения/изменения/удаления (CRUD), которые поддерживает `CrudRepository`, интерфейс `UserRepository` определяет метод `findByUsername()`, который мы будем использовать для поиска учетной записи по имени пользователя.

Как рассказывалось в главе 3, Spring Data JPA автоматически генерирует реализацию этого интерфейса во время выполнения. Таким образом, теперь мы готовы написать свою службу хранения учетных записей, использующую этот репозиторий.

Создание службы хранения учетных записей

Как вы наверняка помните, интерфейс `UserDetailsService` определяет только один метод: `loadUserByUsername()`. То есть это функциональный интерфейс, и его можно реализовать как лямбда-функцию. Поскольку нам нужно, чтобы наша реализация `UserDetailsService` делегировала выполнение операций репозиторию `UserRepository`, ее можно объявить как bean-компонент, используя следующий метод (листинг 5.4).

Листинг 5.4 Определение bean-компонента службы хранения учетных записей

```
@Bean
public UserDetailsService userDetailsService(UserRepository userRepo) {
    return username -> {
        User user = userRepo.findByUsername(username);
        if (user != null) return user;

        throw new UsernameNotFoundException("User '" + username + "' not found");
    };
}
```

Метод `userDetailsService()` принимает параметр `UserRepository`. Чтобы создать bean-компонент, он возвращает лямбда-функцию, которая принимает параметр `username` и использует его для вызова метода `findByUsername()` репозитория `UserRepository`.

В отношении метода `loadByUsername()` действует одно простое правило: он никогда не должен возвращать `null`. Следовательно, если вызов `findByUsername()` возвращает `null`, то лямбда-функция генерирует исключение `UsernameNotFoundException` (которое определено в Spring Security). В противном случае возвращается найденная учетная запись.

Теперь, определив службу хранения учетных записей, которая читает информацию о пользователях из репозитория JPA, нам нужно

реализовать процедуру сохранения новых учетных записей в базе данных. Для этого мы должны создать страницу регистрации для клиентов Taco Cloud.

Регистрация пользователей

Spring Security реализует самые разные аспекты управления безопасностью, но на самом деле эта библиотека не участвует напрямую в процессе регистрации пользователей, поэтому для решения этой задачи нам придется положиться на Spring MVC. Класс `RegistrationController` в листинге 5.5 представляет и обрабатывает форму регистрации.

Листинг 5.5 Контроллер регистрации нового пользователя

```
package tacos.security;

import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import tacos.data.UserRepository;

@Controller
@RequestMapping("/register")
public class RegistrationController {

    private UserRepository userRepo;
    private PasswordEncoder passwordEncoder;

    public RegistrationController(
        UserRepository userRepo, PasswordEncoder passwordEncoder) {
        this.userRepo = userRepo;
        this.passwordEncoder = passwordEncoder;
    }

    @GetMapping
    public String registerForm() {
        return "registration";
    }

    @PostMapping
    public String processRegistration(RegistrationForm form) {
        userRepo.save(form.toUser(passwordEncoder));
        return "redirect:/login";
    }
}
```

Так же как любой типичный контроллер Spring MVC, `RegistrationController` отмечен аннотацией `@Controller`, обозначающей его как контроллер и открывающей его для механизма сканирования компо-

нентов. Он также имеет аннотацию `@RequestMapping`, согласно которой будет обрабатывать запросы с путем `/register`.

В частности, запрос GET с путем `/register` будет обрабатываться методом `registerForm()`, который просто возвращает логическое имя представления `registration`. В листинге 5.6 показан шаблон Thymeleaf представления `registration`.

Листинг 5.6 Шаблон Thymeleaf представления registration

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Taco Cloud</title>
    </head>

    <body>
        <h1>Register</h1>

        <form method="POST" th:action="@{/register}" id="registerForm">

            <label for="username">Username: </label>
            <input type="text" name="username"/><br/>

            <label for="password">Password: </label>
            <input type="password" name="password"/><br/>

            <label for="confirm">Confirm password: </label>
            <input type="password" name="confirm"/><br/>

            <label for="fullname">Full name: </label>
            <input type="text" name="fullname"/><br/>

            <label for="street">Street: </label>
            <input type="text" name="street"/><br/>

            <label for="city">City: </label>
            <input type="text" name="city"/><br/>

            <label for="state">State: </label>
            <input type="text" name="state"/><br/>

            <label for="zip">Zip: </label>
            <input type="text" name="zip"/><br/>

            <label for="phone">Phone: </label>
            <input type="text" name="phone"/><br/>

            <input type="submit" value="Register"/>
        </form>
    </body>
</html>
```

Метод `processRegistration()` обрабатывает форму, отправленную HTTPS-запросом POST. Поля формы связываются с объектом `RegistrationForm` средствами Spring MVC и передаются в вызов метода `processRegistration()` для обработки. Вот как выглядит класс `RegistrationForm`:

```
package tacos.security;

import org.springframework.security.crypto.password.PasswordEncoder;
import lombok.Data;
import tacos.User;

@Data
public class RegistrationForm {

    private String username;
    private String password;
    private String fullname;
    private String street;
    private String city;
    private String state;
    private String zip;
    private String phone;

    public User toUser(PasswordEncoder passwordEncoder) {
        return new User(
            username, passwordEncoder.encode(password),
            fullname, street, city, state, zip, phone);
    }
}
```

По сути, `RegistrationForm` – это просто класс, снабженный аннотациями Lombok и имеющий несколько свойств. Метод `toUser()` использует эти свойства для создания нового объекта `User`, который будет сохраняться `processRegistration()` в `UserRepository`.

Вы, несомненно, заметили, что в `RegistrationController` внедряется компонент `PasswordEncoder`. Это тот самый bean-компонент `PasswordEncoder`, который мы объявили выше. При обработке формы `RegistrationController` передает его методу `toUser()` для шифрования пароля перед сохранением в базу данных. То есть отправленный пароль сохраняется в зашифрованной форме, и служба управления учетными записями пользователей сможет аутентифицировать пользователя по этому зашифрованному паролю.

Теперь приложение Taco Cloud имеет полноценную поддержку регистрации и аутентификации пользователей. Но если запустить приложение прямо сейчас, то вы заметите, что при попытке открыть страницу регистрации вы попадаете на страницу входа. Это обусловлено тем, что по умолчанию все запросы требуют аутентификации. Давайте посмотрим, как перехватываются и защищаются веб-запросы, чтобы исправить эту странную ситуацию с курицей и яйцом.

5.3 Защита веб-запросов

Согласно требованиям безопасности Taco Cloud, пользователь должен аутентифицироваться перед созданием рецепта тако или заказа. Но домашняя страница, страница входа и страница регистрации должны быть доступны для неавторизованных пользователей.

Чтобы организовать такую возможность, нужно объявить bean-компонент `SecurityFilterChain`. Следующий метод с аннотацией `@Bean` показывает минимальное (но бесполезное) объявление компонента `SecurityFilterChain`:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http.build();
}
```

Метод `filterChain()` принимает объект `HttpSecurity`, который действует как построитель, который можно использовать для настройки работы системы безопасности на веб-уровне. После настройки конфигурации безопасности с помощью объекта `HttpSecurity` вызов `build()` создаст компонент `SecurityFilterChain`.

Помимо всего прочего, с помощью `HttpSecurity` можно:

- потребовать выполнения определенных условий безопасности перед обслуживанием запроса;
- отправить пользователю свою страницу входа;
- предоставить пользователям возможности выйти из приложения;
- настроить защиту от подделки межсайтовых запросов.

Перехват запросов с целью убедиться, что пользователь обладает необходимыми привилегиями, является одним из наиболее распространенных подходов, для которых вы будете настраивать `HttpSecurity`. Давайте позаботимся о соответствии Taco Cloud этим требованиям.

5.3.1 Защита запросов

Мы должны убедиться, что запросы с путями `/design` и `/orders` будут обрабатываться, только если они отправлены аутентифицированными пользователями; все другие запросы должны обрабатываться независимо от факта аутентификации. Именно это обеспечивает следующая конфигурация:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
        .authorizeRequests()
        .antMatchers("/design", "/orders").hasRole("USER")
        .antMatchers("/", "/**").permitAll()
```

```

.and()
.build();
}

```

Вызов `authorizeRequests()` возвращает объект `ExpressionUrlAuthorizationConfigurer.ExpressionInterceptUrlRegistry`, с помощью которого можно задать пути и шаблоны URL, а также требования безопасности для этих путей. В данном случае мы задали следующие два правила безопасности:

- запросы с путями `/design` и `/orders` должны обрабатываться, только если они отправлены пользователями с подтвержденными привилегиями `ROLE_USER`. Не включайте префикс `ROLE_` в названия ролей, передаваемых в вызов `hasRole()`, – он подразумевается по умолчанию;
- все остальные запросы должны обрабатываться безоговорочно.

Порядок следования правил имеет значение. Правила безопасности, объявленные первыми, имеют приоритет над объявленными ниже. Если поменять порядок этих двух правил, то ко всем запросам применялась бы функция `allowAll()` – правило для запросов с путями `/design` и `/orders` просто не действовало бы.

Методы `hasRole()` и `allowAll()` – это лишь два из множества методов, используемых для объявления требований безопасности. Полный перечень этих методов приводится в табл. 5.1.

Таблица 5.1 Методы конфигурации, определяющие правила защиты путей

Метод	Описание
<code>access(String)</code>	Разрешает доступ, если выражение на языке Spring Expression Language (SpEL) дает в результате <code>true</code>
<code>anonymous()</code>	Разрешает доступ анонимным пользователям
<code>authenticated()</code>	Разрешает доступ аутентифицированным пользователям
<code>denyAll()</code>	Запрещает доступ без всяких исключений
<code>fullyAuthenticated()</code>	Разрешает доступ, если пользователь полностью аутентифицирован (не был запомнен с помощью функции «запомнить меня»)
<code>hasAnyAuthority(String...)</code>	Разрешает доступ, если пользователь обладает любой из перечисленных привилегий
<code>hasAnyRole(String...)</code>	Разрешает доступ, если пользователь обладает любой из перечисленных ролей
<code>hasAuthority(String)</code>	Разрешает доступ, если пользователь обладает указанной привилегией
<code>hasIpAddress(String)</code>	Разрешает доступ, если запрос получен с указанного IP-адреса
<code>hasRole(String)</code>	Разрешает доступ, если пользователь обладает указанной ролью
<code>not()</code>	Инвертирует значение, возвращаемое предыдущим методом в цепочке
<code>permitAll()</code>	Разрешает доступ всем без всяких условий
<code>rememberMe()</code>	Разрешает доступ пользователям, аутентифицированным с помощью функции «запомнить меня»

Большинство методов в табл. 5.1 определяют базовые правила безопасности для обработки запросов, но они поддерживают только правила, соответствующие их определениям. Для реализации более

гибких правил можно использовать метод `access()`, позволяющий передавать выражения на языке SpEL, определяющие более сложные правила. Spring Security расширяет синтаксис SpEL несколькими значениями и функциями, связанными с безопасностью, которые перечислены в табл. 5.2.

Таблица 5.2 Расширения Spring Security в языке Spring Expression Language

Метод	Описание
<code>authentication</code>	Объект аутентификации пользователя
<code>denyAll</code>	Всегда возвращает <code>false</code>
<code>hasAnyAuthority(String... authorities)</code>	Возвращает <code>true</code> , если пользователь обладает любой из перечисленных привилегий
<code>hasAnyRole(String... roles)</code>	Возвращает <code>true</code> , если пользователь обладает любой из перечисленных ролей
<code>hasAuthority(String authority)</code>	Возвращает <code>true</code> , если пользователь обладает указанной привилегией
<code>hasPermission(Object target, Object permission)</code>	Возвращает <code>true</code> , если пользователь имеет разрешение <code>permission</code> на доступ к объекту <code>targetId</code>
<code>hasPermission(Serializable targetId, String targetType, Object permission)</code>	Возвращает <code>true</code> , если пользователь имеет разрешение <code>permission</code> на доступ к объекту <code>targetId</code> типа <code>targetType</code>
<code>hasRole(String role)</code>	Возвращает <code>true</code> , если пользователь обладает указанной ролью
<code>hasIpAddress(String ipAddress)</code>	Возвращает <code>true</code> , если запрос получен с указанного IP-адреса
<code>isAnonymous()</code>	Возвращает <code>true</code> , если запрос получен от анонимного пользователя
<code>isAuthenticated()</code>	Возвращает <code>true</code> , если запрос получен от аутентифицированного пользователя
<code>isFullyAuthenticated()</code>	Возвращает <code>true</code> , если пользователь полностью аутентифицирован (кроме пользователей, аутентифицированных с помощью функции «запомнить меня»)
<code>isRememberMe()</code>	Возвращает <code>true</code> , если пользователь аутентифицирован с помощью функции «запомнить меня»
<code>permitAll</code>	Всегда возвращает <code>true</code>
<code>principal</code>	Основной объект, представляющий пользователя

Как видите, большинство расширений в табл. 5.2 соответствуют аналогичным методам в табл. 5.1. На самом деле, используя метод `access()` с выражениями `hasRole` и `PermitAll`, можно переписать конфигурацию `SecurityFilterChain`, как показано ниже.

```

@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
        .authorizeRequests()
        .antMatchers("/design", "/orders").access("hasRole('USER')")
        .antMatchers("/", "/**").access("permitAll()")
        .and()
        .build();
}

```

На первый взгляд ничего особенного. В конце концов, эти выражения лишь отражают то, что мы делали выше с использованием методов. Но выражения могут быть гораздо более гибкими. Например, представьте, что (по какой-то безумной причине) нам понадобилось разрешить создавать новые только пользователям с полномочиями ROLE_USER и только по вторникам. В таком случае мы могли бы переписать выражение, как показано в следующей ниже версии метода bean-компоненты SecurityFilterChain:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
        .authorizeRequests()
        .antMatchers("/design", "/orders")
        .access("hasRole('USER') && " +
            "T(java.util.Calendar).getInstance().get(" +
            "T(java.util.Calendar).DAY_OF_WEEK) == " +
            "T(java.util.Calendar).TUESDAY")
        .antMatchers("/", "/**").access("permitAll")
        .and()
        .build();
}
```

Язык SpEL обеспечивает практически неограниченные возможности настройки системы безопасности. Могу поспорить, что вы уже начали придумывать свои интересные правила на основе SpEL.

Потребности приложения Taco Cloud в авторизации легко можно удовлетворить с помощью access() и простых выражений на SpEL. А теперь давайте посмотрим, как настроить страницу входа в систему, чтобы она соответствовала внешнему виду приложения Taco Cloud.

5.3.2 Создание страницы входа

Страница входа по умолчанию выглядит намного лучше, чем простенький диалог HTTP, с которого мы начали, но она все еще остается слишком простой и не соответствует внешнему виду остального приложения Taco Cloud.

Чтобы заменить встроенную страницу входа, сначала нужно сообщить Spring Security путь к этой странице. Это можно сделать вызовом метода formLogin() объекта HttpSecurity, как показано ниже:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    return http
        .authorizeRequests()
        .antMatchers("/design", "/orders").access("hasRole('USER')")
        .antMatchers("/", "/**").access("permitAll()")
        .and()
        .formLogin()
        .loginPage("/login")
```

```

    .and()
    .build();
}

```

Обратите внимание, что перед вызовом `formLogin()` мы связали этот и предыдущий разделы конфигурации вызовом `and()`. Метод `and()` означает, что мы закончили настройку авторизации и готовы применить некоторую дополнительную настройку HTTP. Мы будем использовать `and()` всякий раз, начиная новый раздел конфигурации.

Вслед за методом `and()` вызывается `formLogin()`, начинающий настройку формы входа. Следующий далее вызов `loginPage()` определяет путь к нашей странице входа. Когда библиотека Spring Security обнаружит, что пользователь не прошел аутентификацию, она перенаправит его на эту страницу.

Теперь нам нужно реализовать контроллер, обрабатывающий запросы с этим путем. Поскольку наша страница входа довольно простая – она не содержит ничего, кроме представления, – этот контроллер можно просто объявить контроллером представления в `WebConfig`. Следующий метод `addViewControllers()` настраивает контроллер представления страницы входа вместе с контроллером представления домашней страницы «/»:

```

@Override
public void addViewControllers(ViewControllerRegistry registry) {
    registry.addViewController("/").setViewName("home");
    registry.addViewController("/login");
}

```

Наконец, нам нужно определить саму страницу входа. Поскольку мы используем механизм шаблонов Thymeleaf, следующего шаблона будет вполне достаточно:

```

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Taco Cloud</title>
    </head>

    <body>
        <h1>Login</h1>
        

        <div th:if="${error}">
            Unable to login. Check your username and password.
        </div>

        <p>New here? Click
            <a th:href="@{/register}">here</a> to register.</p>

        <form method="POST" th:action="@{/login}" id="loginForm">
            <label for="username">Username: </label>

```

```

<input type="text" name="username" id="username" /><br/>

<label for="password">Password: </label>
<input type="password" name="password" id="password" /><br/>

<input type="submit" value="Login"/>
</form>
</body>
</html>

```

Ключевыми аспектами, на которые следует обратить внимание в этой странице входа, являются путь и имена полей ввода имени пользователя и пароля. По умолчанию Spring Security ожидает, что запросы на вход будут иметь путь `/login` и содержать поля `username` и `password` с именем пользователя и паролем соответственно. Однако эти настройки можно изменить. Например, следующая конфигурация определяет другие путь и имена полей:

```

.and()
.formLogin()
.loginPage("/login")
.loginProcessingUrl("/authenticate")
.usernameParameter("user")
.passwordParameter("pwd")

```

Здесь мы указали, что запросы на вход будут иметь путь `/authenticate`, а имя пользователя и пароль будут передаваться в полях `user` и `pwd`.

По умолчанию после успешного входа пользователь будет направляться непосредственно на страницу, которую он пытался открыть, когда Spring Security определила, что ему необходимо пройти процедуру аутентификации. Если пользователь изначально запросил страницу входа, то после успешной аутентификации он будет перенаправлен по корневому пути (например, на главную страницу приложения). Однако этот аспект можно изменить, указав страницу по умолчанию для перенаправления после успешного входа, как показано ниже:

```

.and()
.formLogin()
.loginPage("/login")
.defaultSuccessUrl("/design")

```

Согласно этим настройкам, если пользователь напрямую открыл страницу входа и успешно прошел аутентификацию, то он будет перенаправлен на страницу `/design`.

При желании можно принудительно перенаправить пользователя на страницу `/design` после входа, даже если он изначально пытался открыть другую страницу, передав `true` во втором параметре `defaultSuccessUrl`:

```
.and()
.formLogin()
.loginPage("/login")
.defaultSuccessUrl("/design", true)
```

Вход с именем пользователя и паролем – наиболее распространенный способ аутентификации в веб-приложениях. Но давайте рассмотрим другой способ аутентификации с использованием сторонней страницы входа.

5.3.3 Использование сторонних систем аутентификации

Многим из нас доводилось видеть ссылки или кнопки на веб-сайтах с текстом «Войти через Facebook», «Войти через Twitter» или что-то подобное. Вместо просьбы ввести учетные данные на странице входа, специфичные для веб-сайта, они предлагают аутентифицироваться через другой веб-сайт, такой как Facebook, где пользователь, возможно, уже выполнил вход.

Этот тип аутентификации основан на OAuth2 или OpenID Connect (OIDC). OAuth2 – это спецификация авторизации, которую мы будем рассматривать в главе 8 и использовать для защиты REST API, но ее также можно использовать для аутентификации через сторонний веб-сайт. OpenID Connect – это еще одна спецификация безопасности, основанная на OAuth2 и предназначенная для формализации взаимодействий, происходящих в ходе сторонней аутентификации.

Чтобы использовать этот тип аутентификации в нашем приложении Spring, необходимо добавить в спецификацию сборки начальную зависимость от клиента OAuth2:

```
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

Затем нам нужно определить настройки для соединения с одним или несколькими серверами OAuth2 или OpenID Connect, через которые пользователь сможет проходить аутентификацию. По умолчанию Spring Security поддерживает вход через Facebook, Google, GitHub и Okta, но можно настроить и другие серверы, указав несколько дополнительных свойств.

Общий набор свойств, которые необходимо настроить, чтобы ваше приложение действовало как клиент OAuth2/OpenID Connect, выглядит следующим образом:

```
spring:
  security:
    oauth2:
      client:
        registration:
```

```
<oauth2 or openid provider name>:
  clientId: <client id>
  clientSecret: <client secret>
  scope: <comma-separated list of requested scopes>
```

Например, предположим, что мы решили реализовать в Taco Cloud поддержку входа через Facebook. Следующая конфигурация клиента OAuth2 в *application.yml* обеспечит такую возможность:

```
spring:
  security:
    oauth2:
      client:
        registration:
          facebook:
            clientId: <facebook client id>
            clientSecret: <facebook client secret>
            scope: email, public_profile
```

Идентификатор клиента и секретный ключ – это учетные данные, которые идентифицируют ваше приложение в Facebook. Получить идентификатор клиента и секретный ключ можно, создав учетную запись приложения на странице <https://developers.facebook.com/>. Свойство *scope* определяет уровень привилегий, который будет предоставлен приложению. В данном случае приложение будет иметь доступ к адресу электронной почты пользователя и важной информации из его общедоступного профиля Facebook.

В таком простом приложении, как наше, этого более чем достаточно. Когда пользователь попытается открыть страницу, требующую аутентификации, его браузер будет переадресован на Facebook. Если он еще не вошел в Facebook, его встретит страница входа в Facebook. После входа в Facebook ему будет предложено авторизовать ваше приложение и предоставить запрошенную информацию. После этого пользователь будет перенаправлен обратно в приложение, где он будет считаться аутентифицированным.

Однако если безопасность настроена объявлением bean-компоненты *SecurityFilterChain*, то необходимо включить вход через OAuth2 вместе с остальной частью конфигурации:

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
  return http
    .authorizeRequests()
    .mvcMatchers("/design", "/orders").hasRole("USER")
    .anyRequest().permitAll()

    .and()
    .formLogin()
    .loginPage("/login")

    .and()
```

```
.oauth2Login()  
...  
.and()  
.build();  
}
```

Иногда бывает желательно дать пользователям обе возможности аутентификации – традиционным способом ввода имени пользователя и пароля и через стороннюю службу. В этом случае можно задать страницу входа в конфигурации:

```
.and()  
.oauth2Login()  
.LoginPage("/login")
```

Тогда пользователи всегда будут направляться на страницу входа в систему, предоставленную приложением, где они смогут войти со своим именем пользователя и паролем, как обычно, или щелкнуть на ссылке на той же странице и выполнить вход через Facebook. Вот как может выглядеть такая ссылка в HTML-шаблоне страницы:

```
<a th:href="/oauth2/authorization/facebook">Sign in with Facebook</a>
```

Теперь, разобравшись с входом в приложение, рассмотрим аутентификацию с другой стороны: как дать возможность выйти. Выход из приложения так же важен, как вход. Чтобы выполнить выход, нужно просто вызвать метод `logout()` объекта `HttpSecurity`:

```
.and()  
.logout()
```

Он установит фильтр безопасности, который перехватывает запросы POST к `/logout`. Поэтому, чтобы дать возможность выхода из системы, нужно просто добавить форму с кнопкой выхода, как показано ниже:

```
<form method="POST" th:action="@{/logout}">  
  <input type="submit" value="Logout"/>  
</form>
```

Когда пользователь щелкнет на кнопке, его сеанс будет закрыт, и он выйдет из приложения. После этого он будет перенаправлен на страницу входа, где сможет снова войти в систему. Но если вы предпочитаете, чтобы после выхода пользователи перенаправлялись на другую страницу, то можете вызвать `logoutSuccessUrl()` и задать другую целевую страницу:

```
.and()  
.logout()  
.logoutSuccessUrl("")
```

В этом случае после выхода пользователи будут перенаправляться на главную страницу.

5.3.4 Предотвращение подделки межсайтовых запросов

Подделка межсайтовых запросов (Cross-Site Request Forgery, CSRF) – распространенный вид атак на системы безопасности. Такие атаки предполагают предоставление вредоносной веб-страницы пользователю, которая автоматически (и обычно тайно) отправляет заполненную форму другому приложению, причиняя пользователю тот или иной ущерб. Например, на злонамеренном веб-сайте пользователю может быть предложено заполнить форму, которая автоматически отправляет сообщение на URL банковского веб-сайта пользователя (предположительно слабо защищенный и уязвимый для такой атаки) с запросом на перевод денег. Пользователь может даже не подозревать, что произошла атака, пока не заметит пропажу денег со своего счета.

Для защиты от таких атак приложения могут генерировать токен CSRF, помещать этот токен в скрытое поле формы, а затем сохранять его для последующего использования на сервере. Когда пользователь возвращает заполненную форму, вместе с ней на сервер возвращается и токен, который затем сравнивается на сервере с первоначально сгенерированным токеном. Если токен совпадает, обработка запроса продолжается как обычно. В противном случае это может означать, что форма сгенерирована вредоносным веб-сайтом, которому неизвестен токен, сгенерированный сервером.

К счастью, Spring Security имеет встроенную защиту от CSRF. И самое замечательное, что эта защита включена по умолчанию, т. е. ее не требуется настраивать явно. Вам нужно только добавить во все свои формы поле с именем `_csrf`, которое будет хранить токен CSRF.

Spring Security упрощает эту задачу еще больше, помещая токен CSRF в атрибут запроса с именем `_csrf`. Поэтому вы можете скопировать токен CSRF в скрытое поле, как показано в следующем фрагменте шаблона Thymeleaf:

```
<input type="hidden" name="_csrf" th:value="${_csrf.token}"/>
```

Если вы используете библиотеку тегов Spring MVC JSP или Thymeleaf с диалектом Spring Security, то вам даже не нужно явно включать скрытое поле – оно будет добавляться автоматически.

В шаблоне Thymeleaf нужно лишь добавить в элемент `<form>` один из атрибутов с префиксом Thymeleaf. Обычно это не проблема, потому что довольно часто Thymeleaf позволяет отображать путь как контекстно-зависимый. Например, атрибута `th:action`, как показано ниже, вполне достаточно, чтобы механизм шаблонов Thymeleaf включил скрытое поле в форму:

```
<form method="POST" th:action="@{/login}" id="loginForm">
```

Поддержку защиты CSRF можно отключить, но я не хотел бы показывать, как это сделать. Защита CSRF играет важную роль и не требует дополнительных усилий для обработки, поэтому я не вижу никаких причин ее отключать. Однако если вы знаете, что делаете, и вам действительно понадобится отключить эту защиту, то вы сможете сделать это, вызвав `disable()`:

```
.and()  
.csrf()  
.disable()
```

И снова предупреждаю вас: не отключайте защиту CSRF, особенно в промышленных приложениях.

На этом настройку системы безопасности веб-уровня приложения Taco Cloud можно считать законченной. Теперь, помимо всего прочего, у нас есть своя страница входа и поддержка аутентификации пользователей с использованием репозитория JPA, в котором хранятся учетные записи. Далее я покажу, как можно получить информацию о вошедшем пользователе.

5.4 Безопасность на уровне методов

Размышлять о безопасности на уровне веб-запросов легко, но этот уровень не всегда подходит для применения ограничений. Иногда желательно убедиться, что пользователь аутентифицирован и обладает соответствующими привилегиями в точке выполнения защищаемого действия.

Например, представьте, что для нужд администрирования мы реализовали служебный класс с методом, который удаляет все заказы из базы данных. Этот метод, использующий внедренный `OrderRepository`, может выглядеть примерно так:

```
public void deleteAllOrders() {  
    orderRepository.deleteAll();  
}
```

Теперь предположим, что у нас есть контроллер, обрабатывающий запрос POST и вызывающий метод `deleteAllOrders()`:

```
@Controller  
@RequestMapping("/admin")  
public class AdminController {  
  
    private OrderAdminService adminService;  
  
    public AdminController(OrderAdminService adminService) {  
        this.adminService = adminService;  
    }
```

```

    @PostMapping("/deleteOrders")
    public String deleteAllOrders() {
        adminService.deleteAllOrders();
        return "redirect:/admin";
    }
}

```

Мы с легкостью могли бы настроить `SecurityConfig`, чтобы разрешить обработку этого запроса POST, только если он отправлен авторизованным пользователем:

```

.authorizeRequests()
...
.antMatchers(HttpMethod.POST, "/admin/**")
    .access("hasRole('ADMIN')")
...

```

Эти настройки не позволяют никакому неавторизованному пользователю выполнить запрос POST с путем `/admin/deleteOrders` и тем самым удалить все заказы из базы данных.

Но представьте, что в некотором другом контроллере появится метод, вызывающий `deleteAllOrders()`. Вам придется добавить дополнительные проверки, чтобы защитить другие контроллеры.

Однако можно поступить проще – защитить непосредственно метод `deleteAllOrders()`:

```

@PreAuthorize("hasRole('ADMIN')")
public void deleteAllOrders() {
    orderRepository.deleteAll();
}

```

Аннотация `@PreAuthorize` принимает выражение SpEL, и если выражение дает в результате `false`, то метод не вызывается. Если выражение дает в результате `true`, то метод будет вызван. В данном случае `@PreAuthorize` проверяет наличие у пользователя привилегии `ROLE_ADMIN`. Если пользователь обладает этой привилегией, то метод будет вызван и удалит все заказы. В противном случае вызов метода не состоится.

Если `@PreAuthorize` заблокирует вызов, то будет сгенерировано исключение `AccessDeniedException`. Это неконтролируемое исключение, поэтому его не нужно перехватывать, разве что вы решите выполнить какие-то действия для его обработки. Если исключение не будет перехвачено, оно всплывет и в конечном итоге будет перехвачено фильтрами Spring Security и обработано соответствующим образом либо с помощью страницы HTTP 403, либо перенаправлением на страницу входа, если пользователь не был аутентифицирован.

Чтобы получить возможность использовать `@PreAuthorize`, нужно включить глобальную защиту методов. Для этого следует снабдить класс конфигурации безопасности аннотацией `@EnableGlobalMethodSecurity`:

```
@Configuration  
@EnableGlobalMethodSecurity  
public class SecurityConfig extends WebSecurityConfigurerAdapter {  
    ...  
}
```

`@PreAuthorize` – полезная аннотация и может пригодиться во многих случаях, но знайте, что у нее есть ничуть не менее полезный аналог – аннотация `@PostAuthorize`. Аннотация `@PostAuthorize` действует почти так же, как `@PreAuthorize`, но ее выражение вычисляется только после вызова целевого метода. Это позволяет использовать в выражении значение, возвращаемое методом, и учитывать его при принятии решения.

Например, предположим, что у нас есть метод, извлекающий заказ по его идентификатору. Если вы решите ограничить возможность его использования только администраторами или пользователем, создавшим заказ, то можете использовать `@PostAuthorize`, как показано ниже:

```
@PostAuthorize("hasRole('ADMIN') || " +  
    "returnObject.user.username == authentication.name")  
public TacoOrder getOrder(long id) {  
    ...  
}
```

В этом случае `returnObject` – это `TacoOrder`, возвращаемый методом. Если его свойство `user` содержит то же имя пользователя, что хранится в свойстве `authentication.name`, то дальнейшее использование возвращаемого значения будет разрешено. Однако, чтобы выполнить сравнение, нужно вызвать метод и получить объект `TacoOrder`.

Но подождите! Как предотвратить вызов метода, если условие применения безопасности зависит от значения, возвращаемого этим вызовом? Эта проблема решается просто: вызов метода разрешается всегда, а затем, если выражение возвращает `false`, генерируется исключение `AccessDeniedException`.

5.5 Знай своего пользователя

Часто недостаточно просто знать, что пользователь аутентифицирован и обладает некоторыми привилегиями. Нередко бывает желательно знать о них чуть больше, чтобы иметь возможность адаптировать приложение под их привычки и наклонности.

Например, в `OrderController`, в момент создания объекта `TacoOrder`, привязанного к форме заказа, было бы неплохо заполнить поля `TacoOrder` именем и адресом пользователя, чтобы не заставлять его вводить их при создании каждого заказа. Что еще более важно, сохранив заказ, вы должны связать сущность `TacoOrder` с пользователем, создавшим заказ.

Чтобы создать такую связь между сущностью TacoOrder и сущностью User, нужно добавить новое свойство в класс TacoOrder:

```
@Data
@Entity
@Table(name="Taco_Order")
public class TacoOrder implements Serializable {

    ...
    @ManyToOne
    private User user;
    ...
}
```

Аннотация @ManyToOne указывает, что заказ принадлежит одному пользователю и, наоборот, что у пользователя может быть много заказов. (Так как мы используем Lombok, нам не нужно явно определять методы доступа к свойству.)

В OrderController за сохранение заказа отвечает метод processOrder(). Его нужно изменить, чтобы определить аутентифицированного пользователя и вызвать метод setUser() объекта TacoOrder, дабы связать заказ с пользователем.

У нас есть несколько способов определить пользователя. Вот некоторые наиболее распространенные:

- внедрить объект java.security.Principal в метод контроллера;
- внедрить объект org.springframework.security.core.Authentication в метод контроллера;
- использовать org.springframework.security.core.context.SecurityContextHolder, чтобы получить контекст безопасности;
- внедрить параметр метода с аннотацией @AuthenticationPrincipal. (Аннотация @AuthenticationPrincipal реализована в пакете org.springframework.security.core.annotation.)

Вот пример реализации варианта с внедрением в processOrder() параметра java.security.Principal. Используя этот параметр, можно получить учетную запись пользователя из UserRepository:

```
@PostMapping
public String processOrder(@Valid TacoOrder order, Errors errors,
    SessionStatus sessionStatus,
    Principal principal) {

    ...
    User user = userRepository.findByUsername(
        principal.getName());
    order.setUser(user);
    ...
}
```

Это решение прекрасно работает, но захламляет код, реализующий прикладную логику, деталями, имеющими отношение к безопасности. Есть возможность сократить часть кода, связанного с безопасностью, изменив `processOrder()` так, чтобы вместо `Principal` он принимал в параметре объект `Authentication`:

```
@PostMapping  
public String processOrder(@Valid TacoOrder order, Errors errors,  
    SessionStatus sessionStatus,  
    Authentication authentication) {  
  
    ...  
  
    User user = (User) authentication.getPrincipal();  
    order.setUser(user);  
  
    ...  
  
}
```

Имея объект `Authentication`, можно вызвать его метод `getPrincipal()`, чтобы получить объект, представляющий пользователя. Обратите внимание, что `getPrincipal()` возвращает объект типа `java.util.Object`, поэтому его нужно привести к типу `User`.

Однако самое чистое решение – просто принять объект `User` в `processOrder()`, но снабдить его аннотацией `@AuthenticationPrincipal`, как показано ниже:

```
@PostMapping  
public String processOrder(@Valid TacoOrder order, Errors errors,  
    SessionStatus sessionStatus,  
    @AuthenticationPrincipal User user) {  
  
    if (errors.hasErrors()) {  
        return "orderForm";  
    }  
  
    order.setUser(user);  
  
    orderRepo.save(order);  
    sessionStatus.setComplete();  
  
    return "redirect:/";  
}
```

Вся прелесть аннотации `@AuthenticationPrincipal` в том, что она не требует приведения типа (как в случае с `Authentication`) и все детали, касающиеся безопасности, находятся только внутри самой аннотации. Когда вы получите объект `User` в `processOrder()`, он будет готов к использованию и связыванию с `TacoOrder`.

Есть еще один способ получить информацию об аутентифицированном пользователе, но он более запутанный и содержит больше кода, связанного с безопасностью. Суть его состоит в том, чтобы по-

лучить объект `Authentication` из контекста безопасности, а затем запросить объект, представляющий пользователя:

```
Authentication authentication =
    SecurityContextHolder.getContext().getAuthentication();
User user = (User) authentication.getPrincipal();
```

Несмотря на то что это решение изобилует кодом, связанным с безопасностью, у него есть одно преимущество: его можно использовать в любом месте приложения, а не только в методах контроллеров. Это делает его пригодным для использования на более низких уровнях кода.

Итоги

- Механизм автоконфигурации в Spring Security позволяет получить начальный уровень безопасности, но в большинстве приложений требуется явно настраивать безопасность для удовлетворения уникальных требований.
- Сведения о пользователях можно хранить в реляционных базах данных, LDAP и в любых других хранилищах по вашему выбору.
- Spring Security автоматически защищает от CSRF-атак.
- Информацию об аутентифицированном пользователе можно получить через объект `SecurityContext` (возвращаемый из `SecurityContextHolder.getContext()`) или внедрить в контроллеры с помощью `@AuthenticationPrincipal`.

Работа с конфигурацией



В этой главе рассматриваются следующие темы:

- тонкая настройка автоматически сконфигурированных bean-компонентов;
- применение конфигурационных свойств к компонентам приложения;
- профили Spring.

Вы помните появление первого iPhone? Небольшой прямоугольник из металла и стекла никак не соответствовал тому, что в то время было принято называть телефоном. И все же он стал пионером современной эры смартфонов, изменив наши способы общения. Сенсорные телефоны во многих отношениях проще и мощнее своих предшественников – телефонов-раскладушек, получивших большое распространение к тому моменту, когда появился первый iPhone, но в ту пору было трудно представить, как можно использовать для звонков устройство, имеющее всего одну кнопку.

Механизм автоконфигурации Spring Boot в некотором смысле похож на первый iPhone. Он значительно упрощает разработку приложений Spring. Но после десятилетия, в течение которого мы определяли значения свойств в XML-конфигурации Spring и вызывали методы экземпляров bean-компонентов для их настройки, не сразу стало очевидным, как настраивать свойства bean-компонентов в отсутствие явной конфигурации.

К счастью, Spring Boot предоставляет возможность задавать значения свойств прикладных компонентов с помощью конфигурационных свойств. Конфигурационные свойства – это не что иное, как свойства bean-компонентов, снабженные аннотацией `@ConfigurationProperties`. Spring будет внедрять значения в свойства компонентов, выбирая один из нескольких источников, включая системные свойства JVM, аргументы командной строки и переменные окружения. В разделе 6.2 я покажу, как использовать `@ConfigurationProperties` в наших собственных bean-компонентах, но прежде мы посмотрим, как настроить bean-компоненты из самой библиотеки Spring Boot, которые тоже имеют свойства с аннотациями `@ConfigurationProperties`.

В этой главе мы не будем создавать новых функций для приложения Taco Cloud и зайдемся изучением конфигурационных свойств. Новые знания, полученные здесь, пригодятся нам в следующих главах. Начнем с изучения особенностей механизма автоконфигурации, знание которых поможет нам выполнить тонкую настройку компонентов, автоматически конфигурируемых Spring Boot.

6.1 Тонкая настройка автоконфигурации

Прежде чем углубиться в конфигурационные свойства, важно обозначить следующие разные (но взаимосвязанные) типы конфигураций в Spring:

- *связывание компонентов* – конфигурация, объявляющая прикладные компоненты, которые должны быть созданы в контексте приложения Spring, и определяющая порядок их внедрения друг в друга;
- *внедрение свойств* – конфигурация, устанавливающая значения свойств bean-компонентов в контексте приложения Spring.

В Spring XML и Java эти два типа конфигураций часто явно объявляются в одном и том же месте. В конфигурации Java аннотацией `@Bean` определяются методы, создающие экземпляры компонентов и устанавливающие их свойства. Например, рассмотрим следующий метод с аннотацией `@Bean`, который объявляет `DataSource` для встроенной базы данных H2:

```
@Bean
public DataSource dataSource() {
    return new EmbeddedDatabaseBuilder()
        .setType(H2)
        .addScript("taco_schema.sql")
        .addScripts("user_data.sql", "ingredient_data.sql")
        .build();
}
```

Здесь методы `addScript()` и `addScripts()` инициализируют некоторые свойства типа `String` именами сценариев SQL, которые следует применить к базе данных после того, как источник данных будет соз-

дан. Именно так можно настроить bean-компонент `DataSource`, если не использовать Spring Boot, но механизм автоконфигурации делает этот метод совершенно ненужным.

Если зависимость H2 доступна в пути поиска классов (classpath), то Spring Boot автоматически создаст в контексте приложения Spring соответствующий bean-компонент `DataSource`, который применит сценарии `SQL schema.sql` и `data.sql`.

Но как быть, если сценарии SQL хранятся в файлах с другими именами? А что, если нужно применить больше двух сценариев SQL? Вот тут-то на сцену и выходят конфигурационные свойства. Но прежде чем начать использовать конфигурационные свойства, нужно понять, откуда эти свойства берутся.

6.1.1 Абстракция окружения Spring

Абстракция окружения Spring – это универсальное хранилище для любых настраиваемых свойств. Оно абстрагирует происхождение свойств, чтобы компоненты, нуждающиеся в этих свойствах, могли извлекать их непосредственно из Spring. Окружение Spring поддерживает несколько источников свойств, в том числе:

- системные свойства JVM;
- переменные окружения операционной системы;
- аргументы командной строки;
- файлы конфигурационных свойств приложения.

И объединяет эти свойства в единый источник, откуда они могут внедряться в bean-компоненты Spring. На рис. 6.1 показано, как свойства из источников свойств передаются через абстракцию окружения Spring в bean-компоненты Spring.

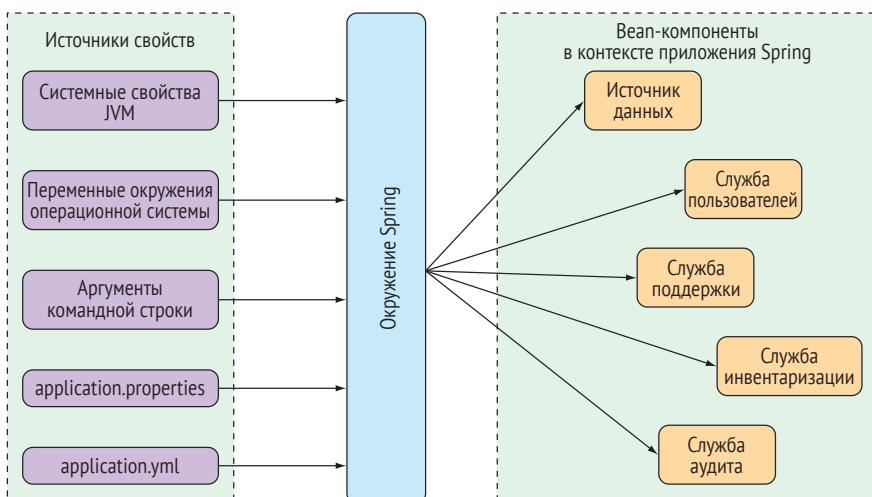


Рис. 6.1 Окружение Spring извлекает свойства из нескольких источников и делает их доступными для bean-компонентов в контексте приложения

Все bean-компоненты, которые автоматически конфигурируются библиотекой Spring Boot, можно настраивать с помощью свойств из окружения Spring. В качестве простого примера предположим, что нам нужно, чтобы базовый контейнер сервлета приложения принимал запросы на каком-либо порту, отличном от порта по умолчанию 8080. Для этого достаточно указать другой порт, установив свойство `server.port` в `src/main/resources/application.properties`:

```
server.port=9090
```

Лично я предпочитаю использовать YAML для настройки конфигурационных свойств. Поэтому я мог бы установить значение `server.port` в `src/main/resources/application.yml` вместо `application.properties`:

```
server:
  port: 9090
```

Если понадобится настроить это свойство извне, то можете указать порт с помощью аргумента командной строки:

```
$ java -jar tacocloud-0.0.5-SNAPSHOT.jar --server.port=9090
```

Если требуется, чтобы приложение всегда прослушивало определенный порт, то его можно установить один раз в переменной окружения операционной системы:

```
$ export SERVER_PORT=9090
```

Обратите внимание, что при определении значений свойств в переменных окружения стиль именования немного отличается, так как приходится учитывать ограничения, накладываемые операционной системой на имена переменных окружения. Но это нормально. Spring в состоянии понять и интерпретировать `SERVER_PORT` как `server.port`.

Как я уже сказал, у нас есть несколько способов установки конфигурационных свойств. На самом деле вы можете использовать несколько сотен конфигурационных свойств для настройки поведения bean-компонентов в Spring. Вы уже видели некоторые из них: `server.port` в этой главе, а также `spring.datasource.name` и `spring.thymeleaf.cache` в предыдущих главах.

В одной главе невозможно охватить все доступные конфигурационные свойства, поэтому мы рассмотрим только некоторые наиболее полезные, с которыми приходится сталкиваться особенно часто. Начнем с нескольких свойств, позволяющих настроить автоматически сконфигурированный источник данных.

6.1.2 Настройка источника данных

На данный момент приложение Taco Cloud все еще не завершено и не готово к развертыванию, но у нас впереди еще несколько глав, где

мы позаботимся об этом. На данный момент в качестве источника данных мы используем встроенную базу данных H2, и она идеально подходит для наших нужд. Но перед развертыванием приложения в промышленном окружении вы, вероятно, захотите рассмотреть более надежное решение для организации базы данных.

Вы можете явно настроить свой собственный компонент `DataSource`, но часто в этом нет необходимости. Вместо этого проще указать URL и учетные данные для доступа к базе данных в конфигурационных свойствах. Например, если вы решите использовать базу данных MySQL, то сможете добавить следующие свойства в `application.yml`:

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword
```

Вам обязательно потребуется добавить в спецификацию сборки соответствующий драйвер JDBC, однако обычно нет необходимости явно указывать класс драйвера JDBC – Spring Boot сможет определить его по структуре URL базы данных. Но если возникнет проблема, то можно попробовать установить свойство `spring.datasource.driver-class-name`:

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword
    driver-class-name: com.mysql.jdbc.Driver
```

Spring Boot использует эти данные при автоматической настройке bean-компоненты `DataSource`. Компонент `DataSource` будет объединен в пул соединений HikariCP, если он доступен в пути к классам. Если нет, то Spring Boot отыщет и использует одну из следующих реализаций пула соединений, присутствующую в пути к классам:

- Tomcat JDBC Connection Pool;
- Apache Commons DBCP2.

Это единственные пулы соединений, доступные механизму автоконфигурации, но вы всегда можете явно настроить bean-компонент `DataSource` и использовать любую другую реализацию пула по вашему выбору.

Выше в этой главе мы предположили, что должен существовать способ, позволяющий указать сценарии инициализации базы данных, которые должны применяться к базе данных при запуске приложения. В этом случае вам пригодятся свойства `spring.datasource.schema` и `spring.datasource.data`:

```
spring:
  datasource:
    schema:
      - order-schema.sql
      - ingredient-schema.sql
      - taco-schema.sql
      - user-schema.sql
    data:
      - ingredients.sql
```

Возможно, явная конфигурация источника данных – не ваш стиль. Вероятно, вы предпочитаете настраивать источники данных в службе имен и каталогов Java (Java Naming and Directory Interface, JNDI; <http://mng.bz/MvEo>) и хотели бы позволить Spring искать настройки оттуда. В этом случае добавьте свойство `spring.datasource.jndi-name`:

```
spring:
  datasource:
    jndi-name: java:/comp/env/jdbc/tacoCloudDS
```

Если вы установите свойство `spring.datasource.jndi-name`, то другие свойства с настройками подключения к источнику данных (если они установлены) будут игнорироваться.

6.1.3 Настройка встроенного сервера

Вы уже видели, как настроить `server.port` – порт контейнера сервлета. Но я не показал, что произойдет, если свойству `server.port` присвоить значение 0, как показано здесь:

```
server:
  port: 0
```

Несмотря на явное назначение порта 0 в `server.port`, сервер не будет прослушивать порт 0. Вместо этого будет выбран случайный номер порта. Это полезно для проведения автоматизированных интеграционных тестов, помогающих убедиться, что любые одновременно выполняемые тесты не конфликтуют с жестко заданным номером порта.

Но базовый сервер – это не только порт. Помимо номера порта необходимо также настроить обработку HTTPS-запросов. Для этого прежде всего нужно создать хранилище ключей с помощью утилиты командной строки `keytool` из комплекта JDK, как показано ниже:

```
$ keytool -keystore mykeys.jks -genkey -alias tomcat -keyalg RSA
```

Вам будет предложено ввести ваше имя и название организации, которые в большинстве случаев не имеют значения. Но при вводе пароля вы должны запомнить его. Ради этого примера я выбрал пароль `letmein`.

Затем вам нужно задать значения нескольких свойств, чтобы включить поддержку HTTPS на встроенном сервере. Все эти настройки

можно указать в командной строке, но это ужасно неудобно. Вместо этого вы, скорее всего, определите их в файле *application.properties* или *application.yml*. В *application.yml* свойства могут выглядеть так:

```
server:  
  port: 8443  
  ssl:  
    key-store: file:///path/to/mykeys.jks  
    key-store-password: letmein  
    key-password: letmein
```

Здесь свойство *server.port* получает значение 8443, обычный выбор для сервера разработки HTTPS. В свойстве *server.ssl.key-store* должен быть прописан путь к файлу, где хранятся ключи. В этом примере используется URL *file://*, определяющий путь к файлу в локальной файловой системе, но если он будет упаковываться в файл приложения JAR, то следует использовать URL *classpath://*. В обоих свойствах, *server.ssl.key-store-password* и *server.ssl.key-password*, задается пароль, который вы ввели при создании хранилища ключей.

При такой настройке свойств приложение должно принимать HTTPS-запросы на порту 8443. В зависимости от типа используемого браузера вы можете увидеть предупреждение о том, что сервер не может подтвердить свою идентичность. Но при работе с локальным сервером разработки это не является причиной для беспокойств.

6.1.4 Настройка журналирования

Большинство приложений предоставляют поддержку журналирования в той или иной форме. И даже если ваше приложение само ничего не журналирует, библиотеки, используемые приложением, обязательно будут журналировать свои действия.

По умолчанию Spring Boot использует механизм журналирования Logback (<http://logback.qos.ch>) и выводит информацию уровня INFO в консоль. Вы, вероятно, уже видели множество записей уровня INFO в журналах, когда запускали наше приложение и другие примеры. Только чтобы напомнить, ниже приводится пример журнала, показывающий его формат по умолчанию (с переносами строк, чтобы уместить их по ширине книжной страницы):

```
2021-07-29 17:24:24.187 INFO 52240 --- [nio-8080-exec-1]  
↳ com.example.demo.Hello                Одна запись в журнале.  
2021-07-29 17:24:24.187 INFO 52240 --- [nio-8080-exec-1]  
↳ com.example.demo.Hello                Другая запись в журнале.  
2021-07-29 17:24:24.187 INFO 52240 --- [nio-8080-exec-1]  
↳ com.example.demo.Hello                Третья запись в журнале.
```

Чтобы иметь максимально полный контроль над настройками журналирования, можно создать файл *logback.xml* в корне пути к классам (в *src/main/resources*). Вот пример простого файла *logback.xml*:

```
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>
                %d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n
            </pattern>
        </encoder>
    </appender>
    <logger name="root" level="INFO"/>
    <root level="INFO">
        <appender-ref ref="STDOUT" />
    </root>
</configuration>
```

С этими настройками те же записи в журнал, что были показаны выше, будут выглядеть следующим образом (с переносами строк, чтобы уместить их по ширине книжной страницы):

```
17:25:09.088 [http-nio-8080-exec-1] INFO com.example.demo.Hello -
Одна запись в журнале.
17:25:09.088 [http-nio-8080-exec-1] INFO com.example.demo.Hello -
Другая запись в журнале.
17:25:09.088 [http-nio-8080-exec-1] INFO com.example.demo.Hello -
Третья запись в журнале.
```

Помимо шаблона журнала, эта конфигурация Logback почти эквивалентна конфигурации по умолчанию, которую вы получите в отсутствие файла *logback.xml*. Но, создав файл *logback.xml*, вы можете получить полный контроль над файлами журналов вашего приложения.

ПРИМЕЧАНИЕ Обсуждение синтаксиса файла *logback.xml* выходит за рамки этой книги. За более полной информацией обращайтесь к документации Logback.

Чаще всего в конфигурации журналирования изменяется уровень фиксируемых сообщений и, возможно, имя файла журнала. Однако эти настройки можно выполнить с помощью свойств конфигурации Spring Boot, не создавая файл *logback.xml*.

Чтобы задать уровень журналирования, нужно определить свойство с префиксом *logging.level*, за которым следует имя механизма журналирования, для которого устанавливается уровень фиксируемых сообщений. Например, предположим, что вы решили задать корневой уровень *WARN*, а для Spring Security – уровень *DEBUG*. Для этого нужно добавить в *application.yml* следующие настройки:

```
logging:
  level:
    root: WARN
    org:
      springframework:
        security: DEBUG
```

Для удобства имя пакета Spring Security можно свернуть в одну строку:

```
logging:  
  level:  
    root: WARN  
    org.springframework.security: DEBUG
```

Теперь предположим, что вы решили назначить файл журнала *TacoCloud.log*, находящийся в */var/logs/*. Для этого нужно настроить свойства *logging.file.path* и *logging.file.name*:

```
logging:  
  file:  
    path: /var/logs/  
    file: TacoCloud.log  
  level:  
    root: WARN  
    org:  
      springframework:  
        security: DEBUG
```

При наличии у приложения разрешения на запись в */var/logs/* журналирование будет производиться в */var/logs/TacoCloud.log*. По умолчанию ротация файлов журналов производится, как только они достигают размера 10 Мбайт.

6.1.5 Использование специальных значений свойств

При настройке свойств можно использовать не только жестко заданные строковые и числовые значения, но также значения других конфигурационных свойств.

Например, предположим (независимо от причины), что вы решили присвоить свойству *greeting.welcome* значение другого свойства – *spring.application.name*. Для этого можно использовать маркеры-заполнители *\${}:*

```
greeting:  
  welcome: ${spring.application.name}
```

Маркеры-заполнители можно даже вставлять в обычный текст:

```
greeting:  
  welcome: You are using ${spring.application.name}.
```

Как видите, настройка собственных компонентов Spring с помощью конфигурационных свойств упрощает внедрение значений в свойства этих компонентов и точную настройку автоконфигурации. Конфигурационные свойства не являются некой особенностью, присущей только bean-компонентам, создаваемым фреймворком Spring. Приложив небольшие усилия, можно воспользоваться преимуществами

вами конфигурационных свойств для настройки прикладных bean-компонентов. Давайте посмотрим, как это сделать.

6.2 Создание своих конфигурационных свойств

Как упоминалось выше, конфигурационные свойства – это не что иное, как свойства bean-компонентов, значения для которых извлекаются из абстракции окружения Spring. Однако я пока не объяснил, как происходит настройка свойств этих bean-компонентов.

Для поддержки внедрения конфигурационных свойств Spring Boot предоставляет аннотацию `@ConfigurationProperties`. Она указывает, что значения могут внедряться в свойства bean-компонента из свойств окружения Spring.

Чтобы продемонстрировать работу `@ConfigurationProperties`, предположим, что мы добавили следующий метод в `OrderController` для вывода списка заказов, сделанных аутентифицированным пользователем:

```
@GetMapping
public String ordersForUser(
    @AuthenticationPrincipal User user, Model model) {

    model.addAttribute("orders",
        orderRepo.findByUserOrderByPlacedAtDesc(user));

    return "orderList";
}
```

Также мы добавили в `OrderRepository` следующий необходимый метод `findByUserOrderByPlacedAtDesc()`:

```
List<Order> findByUserOrderByPlacedAtDesc (User user);
```

Обратите внимание, что имя этого метода репозитория содержит предложение `OrderByPlacedAtDesc`. Часть `OrderBy` указывает на свойство, по значению которого будут упорядочиваться результаты, – в данном случае это свойство `placedAt`. Слово `Desc` в конце говорит, что упорядочение будет производиться в порядке убывания. То есть возвращаемый список заказов будет отсортирован в порядке от самого последнего до самого первого.

Очевидно, что этот метод может пригодиться тем пользователям, которые сделали несколько заказов, но для самых заядлых ценителей тако этот список может оказаться весьма длинным. Когда в окне браузера отображается несколько заказов, это одно, а когда таких заказов сотни, это совсем другое. Допустим, мы решили ограничиться отображением 20 последних заказов. В таком случае мы можем изменить `orderForUser()` следующим образом:

```

@GetMapping
public String ordersForUser(
    @AuthenticationPrincipal User user, Model model) {

    Pageable pageable = PageRequest.of(0, 20);
    model.addAttribute("orders",
        orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

    return "orderList";
}

```

а также изменить OrderRepository:

```
List<TacoOrder> findByUserOrderByPlacedAtDesc(
    User user, Pageable pageable);
```

Здесь вы изменили сигнатуру метода `findByUserOrderByPlacedAtDesc()`, чтобы он принимал параметр `Pageable`. `Pageable` дает Spring Data возможность выбирать некоторое подмножество результатов по номеру и размеру страницы. В методе контроллера `ordersForUser()` мы создали объект `PageRequest`, который реализует интерфейс `Pageable` для запроса первой (нулевой) страницы с размером 20, чтобы получить 20 последних заказов, сделанных пользователем.

Этот прием прекрасно работает, но меня немного беспокоит жестко запрограммированный размер страницы. А что, если позже мы решим, что 20 заказов – это слишком много, и захотим уменьшить это количество до 10? Поскольку число жестко зашито в код, нам придется пересобрать и повторно развернуть приложение.

Чтобы этого избежать, можно установить размер страницы с помощью пользовательского конфигурационного свойства. Для этого, во-первых, нужно добавить новое свойство `pageSize` в `OrderController`, а во-вторых, снабдить `OrderController` аннотацией `@ConfigurationProperties`, как показано в листинге 6.1.

```

@Controller
@RequestMapping("/orders")
@SessionAttributes("order")
@ConfigurationProperties(prefix="taco.orders")
public class OrderController {

    private int pageSize = 20;

    public void setPageSize(int pageSize) {
        this.pageSize = pageSize;
    }

    ...

    @GetMapping
    public String ordersForUser(
        @AuthenticationPrincipal User user, Model model) {

```

```

        pageable = PageRequest.of(0, pageSize);
        model.addAttribute("orders",
            orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));
        return "orderList";
    }

}

```

Наиболее существенное изменение в листинге 6.1 – появление аннотации `@ConfigurationProperties`. Ее атрибуту `prefix` присвоено значение `"taco.orders"`, т. е. при настройке свойства `pageSize` будет использоваться конфигурационное свойство с именем `taco.orders.pageSize`.

По умолчанию новое свойство `pageSize` имеет значение 20, но мы легко можем заменить его любым другим значением, установив свойство `taco.orders.pageSize`. Например, вот как можно настроить это свойство в `application.yml`:

```

taco:
  orders:
    pageSize: 10

```

Если понадобится изменить это значение в процессе работы, то это можно сделать быстро, без повторной сборки, и развертывать приложения, просто установив свойство `taco.orders.pageSize` с помощью переменной окружения:

```
$ export TACO_ORDERS_PAGESIZE=10
```

Для настройки размера списка заказов можно использовать любые средства. Далее мы посмотрим, как определять конфигурационные данные в хранителях (`holders`) свойств.

6.2.1 Определение хранителей конфигурационных свойств

Аннотация `@ConfigurationProperties` может применяться не только к контроллерам или любым другим конкретным типам bean-компонентов. На самом деле `@ConfigurationProperties` часто добавляется к bean-компонентам с единственной целью – для хранения конфигурационных данных. Аннотация отделяет детали конфигурации от контроллеров и других прикладных классов, а также упрощает совместное использование общих конфигурационных свойств несколькими bean-компонентами.

Например, свойство `pageSize` можно выделить из `OrderController` в отдельный класс. В листинге 6.2 показан класс `OrderProps`, который используется именно таким образом.

Листинг 6.2 Выделение свойства pageSize в отдельный класс-хранитель

```
package tacos.web;

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

@Component
@ConfigurationProperties(prefix="taco.orders")
@Data
public class OrderProps {

    private int pageSize = 20;

}
```

Так же как в случае с OrderController, свойство pageSize получает значение по умолчанию 20, а класс OrderProps снабжен аннотацией @ConfigurationProperties с атрибутом prefix="taco.orders". Он еще снабжен аннотацией @Component, поэтому механизм сканирования компонентов Spring автоматически обнаружит его и создаст как bean-компонент в контексте приложения Spring. Это важно, потому что следующим шагом будет внедрение bean-компонента OrderProps в OrderController.

В хранителях конфигурационных свойств нет ничего особенного. Это обычные bean-компоненты, свойства которых внедряются из окружения Spring. Их можно внедрить в любой другой компонент, которому нужны эти свойства. Для OrderController это означает удаление свойства pageSize из класса OrderController и внедрение bean-компонента OrderProps, как показано ниже:

```
private OrderProps props;

public OrderController(OrderRepository orderRepo,
                      OrderProps props) {
    this.orderRepo = orderRepo;
    this.props = props;
}

...

@GetMapping
public String ordersForUser(
    @AuthenticationPrincipal User user, Model model) {

    Pageable pageable = PageRequest.of(0, props.getPageSize());
    model.addAttribute("orders",
```

```

        orderRepo.findByUserOrderByPlacedAtDesc(user, pageable));

    return "orderList";
}

```

Теперь OrderController больше не занимается обработкой своих конфигурационных свойств. Это решение упрощает код OrderController и позволяет повторно использовать свойства в OrderProps в любых других bean-компонентах, которым они могут понадобиться. Более того, теперь все конфигурационные свойства, относящиеся к заказам, находятся в одном месте – в классе OrderProps. Если понадобится добавить, удалить, переименовать или как-то иначе изменить свойства, это придется сделать только в OrderProps. А для тестирования легко установить свойства конфигурации непосредственно в экземпляре OrderProps, предназначенном исключительно для тестирования, и передать его контроллеру перед тестом.

Например, представим, что мы используем свойство pageSize в нескольких других bean-компонентах и вдруг решили применить некоторую проверку к этому свойству, чтобы ограничить его значения диапазоном от 5 до 25. Без bean-компонента хранителя нам пришлось бы применить аннотации проверки к OrderController, свойству pageSize и всем другим классам, использующим это свойство. Но поскольку мы выделили pageSize в OrderProps, мы можем внести изменения только в OrderProps:

```

package tacos.web;

import javax.validation.constraints.Max;
import javax.validation.constraints.Min;

import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.stereotype.Component;
import org.springframework.validation.annotation.Validated;

import lombok.Data;

@Component
@ConfigurationProperties(prefix="taco.orders")
@Data
@Validated
public class OrderProps {

    @Min(value=5, message="must be between 5 and 25")
    @Max(value=25, message="must be between 5 and 25")
    private int pageSize = 20;

}

```

Мы точно так же могли бы применить аннотации @Validated, @Min и @Max к OrderController (и любым другим bean-компонентам, в которые внедряется экземпляр OrderProps), но это только загромоздило бы код OrderController. Создав bean-компонент для хранения кон-

фигурационных свойств, мы собрали все нужные свойства в одном месте, оставив относительно чистым код классов, использующих эти свойства.

6.2.2 Объявление метаданных конфигурационного свойства

В зависимости от используемой среды разработки вы могли заметить, что элемент `taco.orders.pageSize` в `application.yml` (или `application.properties`) порождает предупреждение о наличии неизвестного свойства «`taco`». Это предупреждение обусловлено отсутствием метаданных, описывающих только что созданное конфигурационное свойство. На рис. 6.2 показано, как выглядит это предупреждение, которое появляется при наведении указателя мыши на имя свойства в окне Spring Tool Suite.



Рис. 6.2 Предупреждение об отсутствии метаданных, описывающих конфигурационное свойство

Метаданные конфигурационных свойств совершенно необязательны и не мешают использовать эти свойства, но могут пригодиться для получения справки, особенно в среде разработки. Например, при наведении указателя мыши на свойство `spring.security.user.password` я вижу подсказку, как показано на рис. 6.3. Хотя всплывающая справка содержит минимальное описание, ее часто достаточно, чтобы понять, для чего используется свойство и как правильно его использовать.

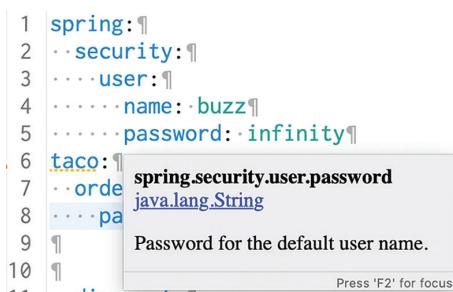


Рис. 6.3 Всплывающая справка с описанием конфигурационного свойства в Spring Tool Suite

Чтобы помочь тем, кто будет использовать конфигурационные свойства, определяемые вами (это может быть вы сами спустя какое-то время), обычно рекомендуется создать некоторые метаданные

с описанием этих свойств. По крайней мере, это избавит от раздражающих желтых предупреждений в среде разработки.

Чтобы определить метаданные для конфигурационных свойств, создайте в папке *META-INF* (например, *src/main/resources/META-INF*) файл с именем *additional-spring-configuration-metadata.json*.

Быстрое добавление отсутствующих метаданных

Если вы используете Spring Tool Suite, то можете быстро добавить отсутствующие метаданные свойств. Наведите указатель мыши на конфигурационное свойство, чтобы появилось всплывающее предупреждение об отсутствии метаданных, и откройте диалог быстрого исправления проблемы, нажав комбинацию **CMD+1** в Mac OS или **Ctrl+1** в Windows и Linux (см. рис. 6.4).

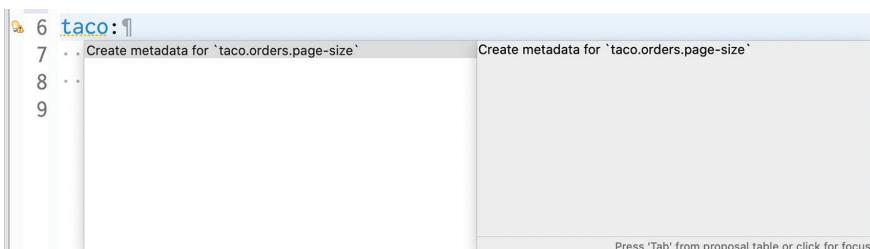


Рис. 6.4 Создание метаданных для конфигурационного свойства с помощью инструмента быстрого исправления ошибок в Spring Tool Suite

Затем выберите пункт **Create Metadata for...** (Создать метаданные для...), чтобы добавить метаданные для свойства. При необходимости этот инструмент сам создаст файл *META-INF/additional-spring-configuration-metadata.json* и заполнит его некоторыми метаданными для свойства *pageSize*, как показано ниже:

```
{"properties": [{"name": "taco.orders.page-size", "type": "java.lang.String", "description": "A description for 'taco.orders.page-size'}]}
```

Обратите внимание, что в метаданных указано имя свойства *taco.orders.page-size*, тогда как фактическое имя свойства – *pageSize*. Гибкость интерпретации имен свойств в Spring Boot допускает различные варианты написания имен свойств, например имя *taco.orders.page-size* эквивалентно *taco.orders.pageSize*, поэтому не имеет большого значения, какую форму использовать.

Первоначальные метаданные, записанные в файл *additional-spring-configuration-metadata.json*, могут служить хорошей основой первое

время, но потом у вас может появиться желание изменить или дополнить их. Прежде всего свойство `pageSize` хранит значение не типа `java.lang.String`, поэтому нужно изменить тип на `java.lang.Integer`. Также желательно изменить метасвойство `description` и дать в нем более точное описание назначения свойства `pageSize`. В следующем примере показано, как могут выглядеть метаданные после внесения нескольких правок:

```
{"properties": [{  
    "name": "taco.orders.page-size",  
    "type": "java.lang.Integer",  
    "description": "Sets the maximum number of orders to display in a list."  
}]}
```

При наличии этих метаданных предупреждения должны исчезнуть. Более того, при наведении указателя мыши на свойство `taco.orders.pageSize` появится описание, показанное на рис. 6.5.

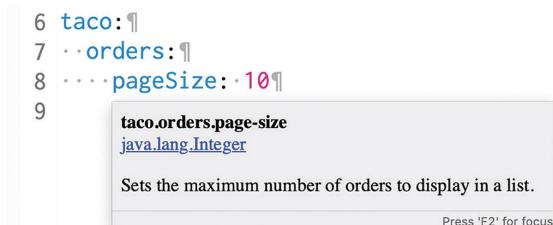


Рис. 6.5 Всплывающая подсказка с описанием конфигурационного свойства

Кроме того, как показано на рис. 6.6, включается поддержка автодополнения в среде разработки, как если бы это свойство было встроенным конфигурационным свойством Spring.

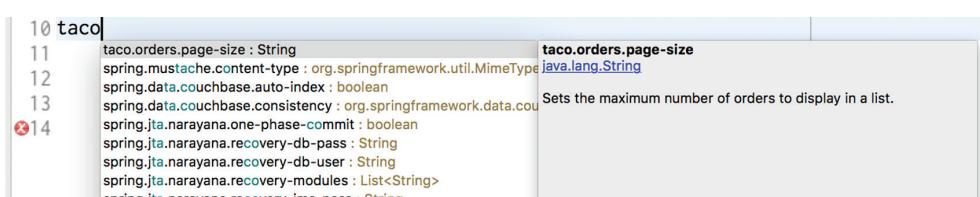


Рис. 6.6 Метаданные с описанием конфигурационного свойства включают поддержку автодополнения

Как видите, конфигурационные свойства помогают настраивать и компоненты механизма автоконфигурации, и ваши собственные прикладные компоненты. А что, если понадобится настроить разные свойства для разных окружений развертывания? Давайте посмотрим, как это организовать с помощью профилей Spring.

6.3 Настстройка с помощью профилей

Когда приложение приходится развертывать в разных окружениях, некоторые их настройки могут отличаться. Например, параметры соединения с базой данных, скорее всего, в окружении разработки будут одними, в окружении тестирования – другими и в промышленном окружении – третьими. Один из способов обеспечить уникальные настройки в разных окружениях – задавать значения конфигурационных свойств в переменных окружения, а не в файлах *application.properties* и *application.yml*.

Например, во время разработки можно использовать встроенную базу данных H2 с автоматической настройкой, а в промышленном окружении задавать настройки базы данных в переменных окружения:

```
% export SPRING_DATASOURCE_URL=jdbc:mysql://localhost/tacocloud  
% export SPRING_DATASOURCE_USERNAME=tacouser  
% export SPRING_DATASOURCE_PASSWORD=tacopassword
```

Это вполне работоспособный подход, но довольно громоздкий, когда требуется определить больше одного-двух конфигурационных свойств в переменных окружения. Кроме того, при таком подходе нет возможности следить за изменениями в переменных окружения или быстро откатывать изменения в случае ошибки.

Поэтому я предпочитаю использовать профили Spring. Профили – это разновидность условной конфигурации, в которой разные bean-компоненты и конфигурационные классы и свойства применяются или игнорируются в зависимости от профиля, действующего во время выполнения.

Например, представьте, что в процессе разработки и отладки мы собирались использовать встроенную базу данных H2 и установить уровень журналирования DEBUG, а в производственном окружении использовать внешнюю базу данных MySQL и уровень журналирования WARN. На этапе разработки можно просто не определять никаких свойств источника данных и получить автоматически настроенную базу данных H2. А уровень журналирования можно установить с помощью свойства *logging.level.tacos* в *application.yml*:

```
logging:  
  level:  
    tacos: DEBUG
```

Это именно то, что нужно для разработки. Но если развернуть это приложение в промышленном окружении без каких-либо изменений в *application.yml*, то уровень журналирования останется равным DEBUG и по-прежнему будет использоваться встроенная база данных H2. Чтобы изменить ситуацию, нужно определить профиль со свойствами, подходящими для промышленного окружения.

6.3.1 Определение свойств профиля

Один из способов определить свойства профиля – создать еще один YAML-файл свойств, содержащий только свойства, имеющие настройки, характерные для промышленного окружения. Имя файла должно выбираться в соответствии со следующим соглашением: *application-{имя профиля}.yml* или *application-{имя профиля}.properties*. Например, можно создать новый файл с именем *application-prod.yml*, содержащий следующие свойства:

```
spring:
  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword

logging:
  level:
    tacos: WARN
```

Другой способ определения свойств профилей основывается исключительно на использовании конфигураций в формате YAML. Он предполагает определение свойств профилей общих свойств в *application.yml* с разделением тремя дефисами и добавлением свойств *spring.profiles* с именами профилей. При таком подходе к определению свойств для промышленного окружения в едином файле *application.yml* содержимое этого файла будет выглядеть так:

```
logging:
  level:
    tacos: DEBUG

---
spring:
  profiles: prod

  datasource:
    url: jdbc:mysql://localhost/tacocloud
    username: tacouser
    password: tacopassword

logging:
  level:
    tacos: WARN
```

Как видите, этот файл *application.yml* разделен на две части тремя дефисами (---). Во втором разделе указывается значение для *spring.profiles*, сообщающее, что следующие свойства применяются к профилю *prod*. С другой стороны, в первом разделе отсутствует определение свойства *spring.profiles*. Соответственно, определяемые здесь

свойства являются общими для всех профилей или используются по умолчанию, если в активном профиле не заданы другие значения.

Независимо от того, какой профиль активен при запуске приложения, уровень журналирования для пакета `tacos` будет установлен равным DEBUG, в соответствии с определением свойства в профиле по умолчанию. Но если активен профиль `rprod`, то свойство `logging.level.tacos` будет переопределено и получит значение WARN. Аналогично, если активен профиль `rprod`, свойства источника данных будут настроены на использование внешней базы данных MySQL.

Вы можете определить свойства для любого количества профилей, создав дополнительные файлы YAML или файлы свойств с именами, следующими шаблону `application-{имя профиля}.yml` или `application-{имя профиля}.properties`. Или, если хотите, добавьте еще три дефиса в `application.yml` с другим именем профиля в свойстве `spring.profiles` и все необходимые свойства профиля. Ни один из этих подходов не имеет никаких преимуществ, однако от себя замечу, что размещение всех свойств профилей в одном файле YAML лучше подходит, когда количество этих свойств невелико, а использование отдельных файлов для профилей лучше подходит, когда имеется большое количество свойств.

6.3.2 Активация профилей

Само определение свойств конкретных профилей бесполезно, если эти профили неактивны. Но как активизировать профиль? Чтобы активизировать профиль, достаточно включить его в список имен профилей, указанный в свойстве `spring.profiles.active`. Это можно сделать, например, в `application.yml`:

```
spring:  
  profiles:  
    active:  
      - rprod
```

Но это, пожалуй, худший из возможных способов активизации профиля. Если указать активный профиль в `application.yml`, то он станет профилем по умолчанию, и вы не получите ни одного из преимуществ использования профилей для настройки конкретных свойств. Вместо этого я рекомендую задавать активные профили с помощью переменных окружения. В промышленном окружении нужно установить переменную `SPRING_PROFILES_ACTIVE`, как показано ниже:

```
% export SPRING_PROFILES_ACTIVE=rprod
```

После этого все приложения, развернутые на этом компьютере, будут использовать активный профиль `rprod`, и соответствующие конфигурационные свойства будут иметь приоритет перед свойствами в профиле по умолчанию.

Если приложение запускается как выполняемый файл JAR, назначить активный профиль можно с помощью параметра командной строки, например:

```
% java -jar taco-cloud.jar --spring.profiles.active=prod
```

Обратите внимание, что имя свойства `spring.profiles.active` содержит слово *profiles* во множественном числе. Это означает, что при необходимости можно указать несколько активных профилей. При назначении активных профилей через переменную окружения они перечисляются через запятую:

```
% export SPRING_PROFILES_ACTIVE=prod,audit,ha
```

Но в файле YAML список должен оформляться так:

```
spring:  
  profiles:  
    active:  
      - prod  
      - audit  
      - ha
```

Также стоит отметить, что при развертывании приложения Spring в Cloud Foundry автоматически активируется профиль с именем `cloud`. Если Cloud Foundry является вашим промышленным окружением, то обязательно определите свойства, характерные для этого окружения, в профиле `cloud`.

Как оказывается, профили могут пригодиться не только для условного назначения конфигурационных свойств в приложениях Spring. Например, есть возможность объявлять bean-компоненты, специфичные для активного профиля.

6.3.3 Условное создание bean-компонентов для профилей

Иногда полезно организовать разные наборы bean-компонентов для разных профилей. Обычно на этапе запуска приложения создаются все bean-компоненты, объявленные в конфигурации Java, независимо от активного профиля. Но представьте, что вам нужно, чтобы некоторые bean-компоненты создавались, только если активен определенный профиль. В таких ситуациях вам поможет аннотация `@Profile`.

Например, предположим, что у нас есть bean-компонент `CommandLineRunner`, объявленный в `TacoCloudApplication`, который заполняет встроенную базу данных информацией об ингредиентах на этапе запуска приложения. Этот компонент помогает во время разработки, но совершенно не нужен (и даже нежелателен) во время нормальной эксплуатации приложения. Чтобы исключить загрузку данных об ин-

тредиентах при каждом запуске приложения в промышленном окружении, можете применить аннотацию `@Profile` к методу компонента `CommandLineRunner`:

```
@Bean
@Profile("dev")
public CommandLineRunner dataLoader(IngredientRepository repo,
    UserRepository userRepo, PasswordEncoder encoder) {
    ...
}
```

Или предположим, что компонент `CommandLineRunner` должен создаваться, только если активен профиль `dev` или `qa`. В таком случае можно перечислить профили, как показано ниже:

```
@Bean
@Profile({"dev", "qa"})
public CommandLineRunner dataLoader(IngredientRepository repo,
    UserRepository userRepo, PasswordEncoder encoder) {
    ...
}
```

Теперь данные об ингредиентах будут загружаться, только если активен профиль `dev` или `qa`. Это означает, что вам придется активировать профиль `dev` при запуске приложения в окружении разработки. Но было бы еще удобнее, если бы bean-компонент `CommandLineRunner` создавался всегда, если неактивен профиль `prod`. В таком случае аннотацию `@Profile` можно применить так:

```
@Bean
@Profile("!prod")
public CommandLineRunner dataLoader(IngredientRepository repo,
    UserRepository userRepo, PasswordEncoder encoder) {
    ...
}
```

Здесь восклицательный знак (!) играет роль отрицания, т. е. в таком виде аннотация говорит, что bean-компонент `CommandLineRunner` будет создан, если профиль `prod` неактивен.

Также аннотацию `@Profile` можно применить ко всему классу, отмеченному аннотацией `@Configuration`. Например, предположим, что нам потребовалось извлечь bean-компонент `CommandLineRunner` в отдельный конфигурационный класс с именем `DevelopmentConfig`. Тогда мы могли бы снабдить `DevelopmentConfig` аннотацией `@Profile`:

```
@Profile({"!prod", "!qa"})
@Configuration
public class DevelopmentConfig {

    @Bean
    public CommandLineRunner dataLoader(IngredientRepository repo,
```

```
UserRepository userRepo, PasswordEncoder encoder) {  
    ...  
}  
}
```

Здесь компонент `CommandLineRunner` (и все другие компоненты, определенные в `DevelopmentConfig`) будет создан, только если неактивны профили `prod` и `qa`.

Итоги

- Компоненты Spring можно аннотировать с помощью `@ConfigurationProperties`, чтобы включить поддержку внедрения значений из нескольких источников свойств.
- Значения конфигурационных свойств можно определять в аргументах командной строки, в переменных окружения, в системных свойствах JVM, в файлах свойств или файлах YAML и в других параметрах.
- Используйте конфигурационные свойства для переопределения значений, присваиваемых механизмом автоконфигурации, таких как URL источника данных или уровень журналирования.
- Профили Spring и источники свойств можно использовать для условной настройки конфигурационных свойств в зависимости от активного профиля (профилей).

Часть II

Интеграция с приложениями Spring

В этой второй части мы посмотрим, как интегрировать наши приложения с другими приложениями Spring.

Глава 7 продолжает обсуждение Spring MVC, начатое в главе 2, и рассматривает приемы разработки REST API в Spring. В ней мы посмотрим, как определять конечные точки REST в Spring MVC, как автоматически генерировать конечные точки REST с помощью Spring Data REST и как использовать REST API. Глава 8 демонстрирует приемы защиты API с помощью Spring Security OAuth 2 и авторизации клиентского кода для доступа к API, защищенным с помощью OAuth 2. В главе 9 рассматривается использование асинхронной связи, позволяющее приложению Spring отправлять и получать сообщения с применением службы сообщений Java (Java Message Service, JMS), RabbitMQ и Kafka. И наконец, в главе 10 обсуждается декларативная интеграция приложений с использованием проекта Spring Integration. В ней мы рассмотрим обработку данных в режиме реального времени, приемы определения потоков интеграции и интеграцию с внешними системами, такими как электронная почта и файловые системы.

7

Создание служб REST

В этой главе рассматриваются следующие темы:

- определение конечных точек REST с помощью Spring MVC;
- автоматическое создание конечных точек REST;
- использование REST API.

«Век веб-браузеров закончился. Что дальше?»

Несколько лет назад я слышал, как кто-то предположил, что век веб-браузеров приближается к концу и их место займет что-то другое. Возможно ли это? Чем можно заменить почти вездесущий веб-браузер? С помощью чего получать услуги от растущего числа сайтов и онлайн-служб, если не веб-браузера? Похоже на бред сумасшедшего!

Вернемся в наши дни, и, как мы видим, веб-браузер никуда не делся. Но он перестал быть основным средством доступа в интернет. Мобильные устройства, планшеты, смарт-часы и голосовые устройства стали обычным явлением. И даже многие браузерные приложения на самом деле являются приложениями на JavaScript, вследствие чего браузер перестал быть простым средством отображения контента, формируемого сервером.

С таким широким выбором вариантов многие приложения пришли к общей архитектуре, согласно которой пользовательский интерфейс реализует клиент, а сервер предоставляет API, через который разные клиенты могут взаимодействовать с серверными функциями.

В этой главе мы с помощью Spring реализуем REST API для приложения Taco Cloud. Мы используем все, что узнали о Spring MVC в гла-

ве 2, и создадим конечные точки RESTful, обслуживаемые контроллерами Spring MVC. Мы также организуем автоматическое создание конечных точек REST для репозиториев Spring Data, которые определили в главах 3 и 4. И в заключение рассмотрим способы тестирования и защиты этих конечных точек.

Но сначала напишем несколько новых контроллеров Spring MVC, реализующих серверные функции и обеспечивающих работу конечных точек REST, которые будут использоваться веб-интерфейсом.

7.1 Создание контроллеров RESTful

REST API мало чем отличаются от веб-сайтов. И те, и другие предполагают отправку ответов на HTTP-запросы. Но, в отличие от веб-сайтов, которые отвечают на запросы отправкой разметки HTML, интерфейсы REST API обычно возвращают ответ в формате, ориентированном на передачу данных, таком как JSON или XML.

В главе 2 вы использовали аннотации `@GetMapping` и `@PostMapping` для извлечения данных, отправленных серверу. Те же самые аннотации можно применять при определении REST API. Но, кроме того, Spring MVC поддерживает еще несколько аннотаций для различных типов HTTP-запросов, перечисленных в табл. 7.1.

Таблица 7.1 Аннотации Spring MVC для обработки HTTP-запросов¹

Аннотация	Метод HTTP	Типичное применение ¹
<code>@GetMapping</code>	Для обработчиков запросов HTTP GET	Чтение данных из ресурса
<code>@PostMapping</code>	Для обработчиков запросов HTTP POST	Создание ресурса
<code>@PutMapping</code>	Для обработчиков запросов HTTP PUT	Изменение содержимого ресурса
<code>@PatchMapping</code>	Для обработчиков запросов HTTP PATCH	Изменение содержимого ресурса
<code>@DeleteMapping</code>	Для обработчиков запросов HTTP DELETE	Удаление ресурса
<code>@RequestMapping</code>	Для универсальных обработчиков запросов; HTTP-метод задается в атрибуте <code>method</code> аннотации	

Чтобы увидеть эти аннотации в действии, создадим простую конечную точку REST, которая извлекает несколько недавно созданных рецептов тако.

7.1.1 Извлечение данных с сервера

Одна из возможностей, которую мы хотели бы реализовать в приложении Taco Cloud, – позволить любителям тако создавать свои рецеп-

¹ Прямое отображение HTTP-методов в операции создания, чтения, изменения и удаления (Create, Read, Update, Delete; CRUD) нельзя назвать идеальным, но на практике именно так они используются чаще всего, и мы тоже будем использовать их так в нашем приложении Taco Cloud.

ты и делиться ими с другими. Для этого можно, например, отображать список тако, недавно созданных на веб-сайте.

Для поддержки этой возможности нужно создать конечную точку, которая обрабатывает запросы GET с путем `/api/tacos`, включающим параметр `recent`, и возвращает в ответ список недавно созданных рецептов тако. Мы создадим новый контроллер, который будет обрабатывать такие запросы. Он показан в листинге 7.1.

Листинг 7.1 Контроллер RESTful для обработки запросов к API создания рецептов тако

```
package tacos.web.api;

import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Sort;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import tacos.Taco;
import tacos.data.TacoRepository;

@RestController
@RequestMapping(path="/api/tacos", produces="application/json")
@CrossOrigin(origins="http://tacocloud:8080")
public class TacoController {

    private TacoRepository tacoRepo;

    public TacoController(TacoRepository tacoRepo) {
        this.tacoRepo = tacoRepo;
    }

    @GetMapping(params="recent")
    public Iterable<Taco> recentTacos() {
        PageRequest page = PageRequest.of(
            0, 12, Sort.by("createdAt").descending());
        return tacoRepo.findAll(page).getContent();
    }
}
```

The code is annotated with several annotations:

- `@RestController`: A bracketed annotation with the text "Обрабатывает запросы с путем /api/tacos".
- `@RequestMapping(path="/api/tacos", produces="application/json")`: A bracketed annotation with the text "Разрешает обработку межсайтowych запросов".
- `@GetMapping(params="recent")`: A bracketed annotation with the text "Извлекает и возвращает последние рецепты тако".

Имя этого контроллера может показаться вам знакомым. В главе 2 мы создали контроллер с похожим именем `DesignTacoController`, который обрабатывал похожие запросы. Тот контроллер предназначался для возврата ответа в формате HTML, а этот новый контроллер `TacoController`, как предполагается, будет служить конечной точкой REST, согласно аннотации `@RestController`.

Аннотация `@RestController` преследует две цели. Во-первых, это стереотипная аннотация, подобно аннотациям `@Controller` и `@Ser-`

vice, которая отмечает класс для обнаружения механизмом сканирования. А во-вторых, что намного важнее для нашего обсуждения REST, `@RestController` сообщает фреймворку Spring, что возвращаемые значения всех методов-обработчиков в контроллере должны включаться непосредственно в тело ответа, а не переноситься в модели в представления для отображения.

С другой стороны, контроллер `TacoController` можно было бы снабдить аннотацией `@Controller`, подобно любому другому контроллеру Spring MVC. Но тогда нам пришлось бы снабдить все методы-обработчики аннотацией `@ResponseBody`, чтобы добиться того же результата. Еще один возможный вариант – возврат объекта `ResponseEntity`, который мы обсудим чуть позже.

Аннотация `@RequestMapping` на уровне класса взаимодействует с аннотацией `@GetMapping` перед методом `recentTacos()`, чтобы указать, что метод `recentTacos()` будет обрабатывать GET-запросы для `/design?recent`.

Обратите внимание, что аннотация `@RequestMapping` устанавливается также атрибут `produces`. Он указывает, что любой метод-обработчик в `TacoController` будет обрабатывать запросы, только если запрос клиента содержит заголовок `Accept` со значением "application/json", и тем самым сообщает, что клиент может обрабатывать ответы только в формате JSON. Такое применение `produces` ограничивает наш API возвратом результатов в формате JSON и позволяет другому контроллеру (возможно, `TacoController` из главы 2) обрабатывать запросы с теми же путями, если эти запросы не требуют возврата результата в формате JSON.

Несмотря на это ограничение (которое, впрочем, вполне согласуется с нашими нуждами), мы можем передать в атрибуте `produces` несколько строк, определяющих типы контента. Например, чтобы разрешить возврат результатов в формате XML, можно добавить "text/xml" в атрибут `produces`:

```
@RequestMapping(path="/api/tacos",
    produces={"application/json", "text/xml"})
```

Еще один важный момент, который можно заметить в листинге 7.1, – аннотация `@CrossOrigin`. Обычно пользовательский интерфейс на основе JavaScript, например, использующий такие фреймворки, как Angular или ReactJS, обслуживается с отдельного хоста и/или порта API (по крайней мере, на данный момент), и веб-браузер не позволит вашему клиенту использовать API. Это ограничение можно обойти, включив заголовки CORS (Cross-Origin Resource Sharing – совместное использование ресурсов между источниками) в ответы сервера. Spring упрощает применение CORS с помощью аннотации `@CrossOrigin`.

В данном случае `@CrossOrigin` позволяет клиентам, загруженным с локального хоста, использовать порт 8080 для доступа к API. Однако в атрибуте `origins` можно передать массив строк, то есть можно указать несколько значений, если понадобится, например:

```

@RestController
@RequestMapping(path="/api/tacos",
    produces="application/json")
@CrossOrigin(origins={"http://tacocloud:8080", "http://tacocloud.com"})
public class TacoController {
    ...
}

```

Метод `recentTacos()` имеет довольно простую логику. Он создает объект `PageRequest`, сообщающий, что нам нужна только первая (0-я) страница с 12 результатами, отсортированными по дате создания рецептов tanto в порядке убывания. Проще говоря, нам нужна дюжина самых последних рецептов. `PageRequest` передается в вызов метода `findAll()` репозитория `TacoRepository`, и затем полученный список возвращается клиенту (который, как вы видели в листинге 7.1, будет использоваться для отображения на стороне пользователя).

Теперь у нас есть начальный Taco Cloud API и мы можем протестировать его с помощью утилиты командной строки `curl` или HTTPie (<https://httpie.org/>). Например, следующая команда показывает, как можно получить список последних созданных рецептов tanto с помощью `curl`:

```
$ curl localhost:8080/api/tacos?recent
```

Аналогичная команда, использующая HTTPie:

```
$ http :8080/api/tacos?recent
```

Изначально база данных пустая, поэтому в ответ на эти запросы будет возвращаться пустой список. Но вскоре мы рассмотрим обработку POST-запросов, сохраняющих рецепты tanto. А пока добавим компонент `CommandLineRunner` для предварительной загрузки в базу данных некоторых тестовых данных. Ниже показано, как можно загрузить несколько ингредиентов и рецептов tanto:

```

@Bean
public CommandLineRunner dataLoader(
    IngredientRepository repo,
    UserRepository userRepo,
    PasswordEncoder encoder,
    TacoRepository tacoRepo) {
    return args -> {
        Ingredient flourTortilla = new Ingredient(
            "FLTO", "Flour Tortilla", Type.WRAP);
        Ingredient cornTortilla = new Ingredient(
            "COTO", "Corn Tortilla", Type.WRAP);
        Ingredient groundBeef = new Ingredient(
            "GRBF", "Ground Beef", Type.PROTEIN);
        Ingredient carnitas = new Ingredient(
            "CARN", "Carnitas", Type.PROTEIN);
        Ingredient tomatoes = new Ingredient(

```

```

        "TMTO", "Diced Tomatoes", Type.VEGGIES);
Ingredient lettuce = new Ingredient(
        "LETC", "Lettuce", Type.VEGGIES);
Ingredient cheddar = new Ingredient(
        "CHED", "Cheddar", Type.CHEESE);
Ingredient jack = new Ingredient(
        "JACK", "Monterrey Jack", Type.CHEESE);
Ingredient salsa = new Ingredient(
        "SLSA", "Salsa", Type.SAUCE);
Ingredient sourCream = new Ingredient(
        "SRCR", "Sour Cream", Type.SAUCE);
repo.save(flourTortilla);
repo.save(cornTortilla);
repo.save(groundBeef);
repo.save(carnitas);
repo.save(tomatoes);
repo.save(lettuce);
repo.save(cheddar);
repo.save(jack);
repo.save(salsa);
repo.save(sourCream);

Taco taco1 = new Taco();
taco1.setName("Carnivore");
taco1.setIngredients(Arrays.asList(
        flourTortilla, groundBeef, carnitas,
        sourCream, salsa, cheddar));
tacoRepo.save(taco1);

Taco taco2 = new Taco();
taco2.setName("Bovine Bounty");
taco2.setIngredients(Arrays.asList(
        cornTortilla, groundBeef, cheddar,
        jack, sourCream));
tacoRepo.save(taco2);

Taco taco3 = new Taco();
taco3.setName("Veg-Out");
taco3.setIngredients(Arrays.asList(
        flourTortilla, cornTortilla, tomatoes,
        lettuce, salsa));
tacoRepo.save(taco3);
};

}

```

Если теперь повторить попытку выполнить запрос с помощью curl или HTTPie к конечной точке /api/tacos?recent, то в ответ вы получите примерно это (отформатировано для удобочитаемости):

```

$ curl localhost:8080/api/tacos?recent
[
{
    "id": 4,

```

```

    "name": "Veg-Out",
    "createdAt": "2021-08-02T00:47:09.624+00:00",
    "ingredients": [
        { "id": "FLTO", "name": "Flour Tortilla", "type": "WRAP" },
        { "id": "COTO", "name": "Corn Tortilla", "type": "WRAP" },
        { "id": "TMT0", "name": "Diced Tomatoes", "type": "VEGGIES" },
        { "id": "LETC", "name": "Lettuce", "type": "VEGGIES" },
        { "id": "SLSA", "name": "Salsa", "type": "SAUCE" }
    ],
},
{
    "id": 3,
    "name": "Bovine Bounty",
    "createdAt": "2021-08-02T00:47:09.621+00:00",
    "ingredients": [
        { "id": "COTO", "name": "Corn Tortilla", "type": "WRAP" },
        { "id": "GRBF", "name": "Ground Beef", "type": "PROTEIN" },
        { "id": "CHED", "name": "Cheddar", "type": "CHEESE" },
        { "id": "JACK", "name": "Monterrey Jack", "type": "CHEESE" },
        { "id": "SRCR", "name": "Sour Cream", "type": "SAUCE" }
    ],
},
{
    "id": 2,
    "name": "Carnivore",
    "createdAt": "2021-08-02T00:47:09.520+00:00",
    "ingredients": [
        { "id": "FLTO", "name": "Flour Tortilla", "type": "WRAP" },
        { "id": "GRBF", "name": "Ground Beef", "type": "PROTEIN" },
        { "id": "CARN", "name": "Carnitas", "type": "PROTEIN" },
        { "id": "SRCR", "name": "Sour Cream", "type": "SAUCE" },
        { "id": "SLSA", "name": "Salsa", "type": "SAUCE" },
        { "id": "CHED", "name": "Cheddar", "type": "CHEESE" }
    ],
}
]

```

Теперь предположим, что мы решили создать конечную точку, которая возвращает рецепт тако по его идентификатору. Используя переменную-заполнитель в пути метода обработчика, можно получить идентификатор и использовать его для поиска объекта Taco:

```

@GetMapping("/{id}")
public Optional<Taco> tacoById(@PathVariable("id") Long id) {
    return tacoRepo.findById(id);
}

```

Поскольку контроллер имеет базовый путь `/api/tacos`, этот метод обрабатывает GET-запросы с путями `/api/tacos/{id}`, где часть пути `{id}` – это переменная-заполнитель. Фактическое значение в запросе присваивается параметру `id`, которое извлекается из заполнителя `{id}` с помощью `@PathVariable`.

Внутри `tacoById()` параметр `id` передается методу `findById()` репозитория для получения Taco. Метод `findById()` возвращает `Optional<Taco>`, потому что есть вероятность, что рецепта тако с указанным идентификатором не существует. Метод контроллера просто возвращает `Optional<Taco>`.

Затем Spring берет `Optional<Taco>` и вызывает его метод `get()`, чтобы сгенерировать ответ. Если в базе данных не нашлось рецепта, соответствующего указанному идентификатору, то тело ответа будет содержать «`null`», а код состояния HTTP-ответа получит значение 200 (OK). Клиент получит ответ, который он не сможет использовать, хотя код состояния подсказывает, что все в порядке. Конечно, лучше было бы вернуть ответ со статусом HTTP 404 (NOT FOUND).

В настоящее время нет простой возможности вернуть код состояния 404 из `tacoById()`. Но если внести несколько небольших изменений, как показано ниже, то можно вернуть нужный код состояния:

```
@GetMapping("/{id}")
public ResponseEntity<Taco> tacoById(@PathVariable("id") Long id) {
    Optional<Taco> optTaco = tacoRepo.findById(id);
    if (optTaco.isPresent()) {
        return new ResponseEntity<>(optTaco.get(), HttpStatus.OK);
    }
    return new ResponseEntity<>(null, HttpStatus.NOT_FOUND);
}
```

Теперь вместо объекта Taco метод `tacoById()` возвращает `ResponseEntity<Taco>`. Если рецепт найден, мы заключаем объект Taco в `ResponseEntity` с HTTP-статусом OK (как и раньше). Но если рецепт тако не найден, мы заключаем `null` в `ResponseEntity` вместе с HTTP-статусом NOT FOUND, чтобы подсказать клиенту, что искомого им рецепта не существует.

Определение конечной точки, возвращающей информацию, – это только начало. А что, если нашему API понадобится получить данные от клиента? Давайте посмотрим, как писать методы контроллера, обрабатывающие входные данные, присылаемые в запросах.

7.1.2 Отправка данных на сервер

На данный момент наш API может возвращать до дюжины последних созданных рецептов тако. Но как эти рецепты создаются?

В настоящий момент можно, конечно, использовать bean-компонент `CommandLineRunner` для предварительной загрузки в базу данных некоторых тестовых данных, но в конечном итоге новые рецепты будут создаваться пользователями. Поэтому нам нужно добавить метод в `TacoController`, обрабатывающий запросы с рецептами тако и сохраняющий их в базе данных. Следующий метод `postTaco()` позволяет контроллеру `TacoController` делать именно это:

```
@PostMapping(consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Taco postTaco(@RequestBody Taco taco) {
    return tacoRepo.save(taco);
}
```

Метод `postTaco()` будет обрабатывать HTTP-запросы POST, поэтому он снабжен аннотацией `@PostMapping`. Мы не указываем здесь атрибут `path`, поэтому метод `postTaco()` будет обрабатывать запросы к конечной точке `/api/tacos`, как указано в аннотации `@RequestMapping` на уровне класса `TacoController`.

Однако мы установили атрибут `consumes`. Атрибут `consumes` определяет формат входных данных в запросе, а `produces` – формат возвращаемых клиенту данных. Здесь мы используем `consumes`, чтобы сказать, что метод будет обрабатывать только запросы с типом содержимого `application/json`.

Параметр `Taco` метода отмечен аннотацией `@RequestBody`, указывающей, что тело запроса должно быть преобразовано в объект `Taco` и присвоено параметру. Эта аннотация играет важную роль – без нее библиотека Spring MVC предположила бы, что объект `Taco` должен извлекаться из параметров запроса (или параметров формы), но аннотация `@RequestBody` требует, чтобы объект `Taco` извлекался из данных в формате JSON, полученных в теле запроса.

Получив объект `Taco`, метод `postTaco()` тут же передает его методу `save()` репозитория `TacoRepository`.

Возможно, вы заметили, что я добавил к методу `postTaco()` аннотацию `@ResponseStatus(HttpStatus.CREATED)`. В нормальных условиях (когда метод не генерирует исключений) все ответы будут получать код состояния HTTP 200 (OK), сообщающий об успешной обработке запроса. Ответ HTTP 200 всегда желателен, но он не всегда достаточно описателен. В случае с запросами POST статус HTTP 201 (CREATED) является более информативным. Он сообщает клиенту, что запрос не только успешно обработан, но и в результате был создан ресурс. Везде, где это уместно, имеет смысл использовать `@ResponseStatus`, чтобы сообщить клиенту наиболее описательный и точный код состояния HTTP.

Для создания нового ресурса `Taco` мы использовали `@PostMapping`, однако POST-запросы можно также использовать для изменения существующего ресурса. Тем не менее запросы POST обычно применяются только для создания ресурсов, а для изменения – запросы PUT и PATCH. Давайте посмотрим, как можно реализовать изменение данных, используя аннотации `@PutMapping` и `@PatchMapping`.

7.1.3 Изменение данных на сервере

Прежде чем писать какой-либо код для обработки HTTP-запросов PUT или PATCH, уделим пару минут размышлению о слоне и слепых мудрецах: почему существуют два разных HTTP-метода для изменения ресурсов?

Это правда, что PUT часто используется для обновления существующих ресурсов, но на самом деле это семантическая противоположность GET. Запросы GET предназначены для передачи данных от сервера клиенту, а запросы PUT – для отправки данных от клиента серверу.

В этом смысле метод PUT предназначен для полной замены ресурса, а не для его изменения. Напротив, целью HTTP-метода PATCH является исправление или частичное изменение ресурса.

Например, представьте, что мы решили организовать возможность изменения адреса в заказе. Один из способов реализовать это в REST API – обработать запрос PUT:

```
@PutMapping(path="/{orderId}", consumes="application/json")
public TacoOrder putOrder(
    @PathVariable("orderId") Long orderId,
    @RequestBody TacoOrder order) {

    order.setId(orderId);
    return repo.save(order);
}
```

Это вполне работоспособное решение, но оно требует, чтобы клиент передал в запросе PUT все данные о заказе. Семантически PUT означает «поместить эти данные по указанному URL», полностью заменив все имеющиеся данные. Если какое-либо из свойств заказа в запросе отсутствует, то оно будет затерто нулевым значением. Даже идентификаторы рецептов тако в заказе нужно будет указать вместе с остальными данными, иначе они будут удалены из заказа.

Итак, PUT выполняет полную замену ресурса. А как можно изменить только часть информации? Вот для этого пригодятся HTTP-запросы PATCH и аннотация `@PatchMapping`. Ниже показано, как может выглядеть метод контроллера, обрабатывающий запросы PATCH для изменения заказа:

```
@PatchMapping(path="/{orderId}", consumes="application/json")
public TacoOrder patchOrder(@PathVariable("orderId") Long orderId,
                           @RequestBody TacoOrder patch) {

    TacoOrder order = repo.findById(orderId).get();
    if (patch.getDeliveryName() != null) {
        order.setDeliveryName(patch.getDeliveryName());
    }
    if (patch.getDeliveryStreet() != null) {
        order.setDeliveryStreet(patch.getDeliveryStreet());
    }
    if (patch.getDeliveryCity() != null) {
        order.setDeliveryCity(patch.getDeliveryCity());
    }
    if (patch.getDeliveryState() != null) {
        order.setDeliveryState(patch.getDeliveryState());
    }
    if (patch.getDeliveryZip() != null) {
```

```
        order.setDeliveryZip(patch.getDeliveryZip());
    }
    if (patch.getCcNumber() != null) {
        order.setCcNumber(patch.getCcNumber());
    }
    if (patch.getCcExpiration() != null) {
        order.setCcExpiration(patch.getCcExpiration());
    }
    if (patch.getCcCVV() != null) {
        order.setCcCVV(patch.getCcCVV());
    }
    return repo.save(order);
}
```

Первое, на что следует обратить внимание, – метод `patchOrder()` отмечен аннотацией `@PatchMapping` вместо `@PutMapping`, т. е. он предназначен для обработки HTTP-запросов PATCH.

И еще что сразу бросается в глаза – метод `patchOrder()` немного сложнее метода `putOrder()`. Это связано с тем, что аннотации сопоставления Spring MVC, включая `@PatchMapping` и `@PutMapping`, определяют конкретный тип запросов, которые должен обрабатывать метод, но они не определяют порядок обработки запросов. Несмотря на то что метод PATCH семантически подразумевает частичное изменение, вы должны написать код, который фактически выполняет такое изменение.

В методе `putOrder()` мы принимаем полный набор данных для заказа и просто сохраняем их, следуя семантике HTTP-метода PUT. Но чтобы `patchMapping()` следовал семантике HTTP-метода PATCH, он должен проверять, какие данные доступны в запросе. Вместо полной замены заказа новыми данными он проверяет каждое поле во входном объекте `TacoOrder` и применяет любые ненулевые значения к существующему заказу. Такой подход позволяет клиенту отправлять только те свойства, которые следует изменить, а серверу – сохранять существующие данные для любых свойств, не указанных клиентом.

Несколько способов обработки запросов PATCH

Подход применения изменений, реализованный в методе `patchOrder()`, имеет следующие ограничения:

- если предполагается, что нулевые значения указывают на отсутствие изменений, то клиент не сможет обнулить выбранное им поле;
- нет возможности удалить или добавить подмножество элементов из коллекции; если клиент захочет добавить или удалить элемент коллекции, он должен отправить полную измененную коллекцию.

На самом деле не существует жесткого правила, регламентирующего обработку запросов PATCH или представление входных данных. Вместо фактических предметных данных клиент может отправить описание применяемых изменений. Обработчик запроса при этом должен правильно интерпретировать такие описания.

Обратите внимание, что путь запроса в `@PutMapping` и `@PatchMapping` ссылается на ресурс, который нужно изменить. Пути обрабатываются точно так же, как в методах с аннотацией `@GetMapping`.

Теперь вы знаете, как получать и сохранять ресурсы с помощью `@GetMapping` и `@PostMapping`. И увидели два способа изменения ресурса с помощью `@PutMapping` и `@PatchMapping`. Осталось только предусмотреть обработку запросов на удаление ресурса.

7.1.4 Удаление данных на сервере

Иногда данные становятся ненужными. В таких случаях клиент должен иметь возможность удалить ресурс, отправив HTTP-запрос `DELETE`.

Для объявления методов, обрабатывающих запросы `DELETE`, используется аннотация `@DeleteMapping` из Spring MVC. Например, представьте, что мы решили реализовать в нашем API возможность удаления ресурса заказа. Для этого можно реализовать такой метод:

```
@DeleteMapping("/{orderId}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteOrder(@PathVariable("orderId") Long orderId) {
    try {
        repo.deleteById(orderId);
    } catch (EmptyResultDataAccessException e) {}
}
```

К этому моменту идея еще одной аннотации отображения должна быть понятна вам без лишних пояснений. Вы уже видели `@GetMapping`, `@PostMapping`, `@PutMapping` и `@PatchMapping`. Каждая из них сообщает, что метод должен обрабатывать HTTP-запросы соответствующего типа. Скорее всего, вас не удивит, что `@DeleteMapping` используется для обозначения того факта, что метод `deleteOrder()` предназначен для обработки запросов `DELETE` к конечной точке `/orders/{orderId}`.

Код внутри метода выполняет фактическую работу по удалению заказа. В данном случае он берет идентификатор заказа, переданный в переменной пути в URL, и передает его методу репозитория `deleteById()`. Если заказ существовал на момент вызова этого метода, он будет удален. Если такого заказа в репозитории нет, то будет сгенерировано исключение `EmptyResultDataAccessException`.

Я решил просто перехватить исключение `EmptyResultDataAccessException` и никак его не обрабатывать. Думаю, что попытка удалить несуществующий ресурс мало чем отличается от удаления существующего ресурса – указанный ресурс не будет существовать после вызова метода. Существовал ли он раньше, не имеет значения. Как вариант можно реализовать `deleteOrder()` так, что он будет возвращать `ResponseEntity` с пустым телом и кодом состояния HTTP 404 (NOT FOUND).

Единственное, на что следует обратить внимание в методе `deleteOrder()`, – это аннотация `@ResponseStatus`, гарантирующая возврат

кода состояния HTTP 204 (NO CONTENT). Бессмысленно передавать клиенту какие-либо данные о ресурсе, который больше не существует, поэтому ответы на запросы DELETE обычно не имеют тела и, следовательно, должны возвращать код состояния HTTP, уведомляющий клиента об отсутствии любого содержимого.

Наш Taco Cloud API начинает обретать форму. Теперь можно заняться разработкой клиента, который будет использовать этот API, отображать доступные ингредиенты, принимать заказы и отображать недавно созданные рецепты тако. Но клиентом REST мы займемся чуть позже, в разделе 7.3, а сейчас исследуем способ автоматического создания конечных точек REST API на основе репозиториев Spring Data.

7.2 Включение услуг на основе данных

Как вы видели в главе 3, Spring Data владеет особым волшебством, автоматически создавая реализации репозитория на основе интерфейсов, которые мы определяли в нашем коде. Но в Spring Data кроется еще одна хитрость, которая может помочь определить API для нашего приложения.

Spring Data REST – еще один член семейства Spring Data, который автоматически создает REST API для репозиториев, созданных с помощью Spring Data. Просто добавляя Spring Data REST в сборку, можно получить API с операциями для каждого интерфейса репозитория, который вы определили.

Чтобы начать использовать Spring Data REST, добавим в спецификацию сборки следующую зависимость:

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

Хотите верьте, хотите нет, но это все, что необходимо, чтобы создать REST API в проекте, который уже использует Spring Data для автоматического создания репозиториев. Простым включением в сборку начальной зависимости Spring Data REST приложение получает возможность использовать автоматически созданный REST API для доступа к любым репозиториям, созданным библиотекой Spring Data (включая Spring Data JPA, Spring Data Mongo и т. д.).

Конечные точки REST, которые создает Spring Data REST, ничуть не уступают (а иногда даже превосходят) конечным точкам, реализованным вручную. А теперь давайте удалим все классы с аннотациями `@RestController`, созданные нами к этому моменту, и затем двинемся дальше.

Чтобы опробовать конечные точки, предоставляемые библиотекой Spring Data REST, запустим приложение и исследуем некоторые URL.

Основываясь на наборе репозиториев, которые мы определили для приложения Taco Cloud, мы можем выполнять запросы GET и получать рецепты тако, ингредиенты, заказы и учетные записи пользователей.

Например, ниже показано, как получить список всех ингредиентов, выполнив запрос GET к конечной точке `/ingredients`. В следующем примере используется утилита curl (вывод сокращен – в нем показан только первый ингредиент):

```
$ curl localhost:8080/ingredients
{
  "_embedded" : {
    "ingredients" : [ {
      "name" : "Flour Tortilla",
      "type" : "WRAP",
      "_links" : {
        "self" : {
          "href" : "http://localhost:8080/ingredients/FLTO"
        },
        "ingredient" : {
          "href" : "http://localhost:8080/ingredients/FLTO"
        }
      },
      ...
    }],
    "_links" : {
      "self" : {
        "href" : "http://localhost:8080/ingredients"
      },
      "profile" : {
        "href" : "http://localhost:8080/profile/ingredients"
      }
    }
  }
}
```

Ух ты! Мы не сделали ничего, кроме добавления зависимости в спецификацию сборки, и получили конечную точку, возвращающую список ингредиентов, при этом возвращаемые ресурсы содержат гиперссылки! Эти гиперссылки являются реализациями гипермедиа как механизма состояния приложения (Hypermedia as Engine of Application State, HATEOAS). Клиент, использующий этот API, может (если понадобится) использовать эти гиперссылки для навигации по API и выполнения следующих запросов.

Проект Spring HATEOAS (<https://spring.io/projects/spring-hateoas>) обеспечивает обобщенную поддержку добавления гипермедиассылок в ответы, возвращаемые контроллерами Spring MVC. Но Spring Data REST автоматически добавляет эти ссылки в ответы, генерируемые API.

За и против использования HATEOAS

Общая идея HATEOAS заключается в возможности перемещаться по API почти так же, как человек перемещается по веб-сайту, – переходя по ссылкам. Вместо реализации всех деталей API в клиенте и создания вручную URL для каждого запроса клиент может выбрать ссылку из списка и использовать ее для выполнения следующего запроса. При таком подходе нет необходимости реализовать в клиенте структуру API, вместо этого можно использовать сам API в качестве дорожной карты.

С другой стороны, гиперссылки добавляют дополнительные данные в полезную нагрузку и несколько увеличивают сложность реализации клиента, требуя, чтобы клиент знал, как перемещаться по гиперссылкам. По этой причине разработчики API часто отказываются от использования HATEOAS, а разработчики клиентов просто игнорируют гиперссылки, если они присутствуют в API.

Помимо наличия гиперссылок, получаемых в ответах Spring Data REST, мы проигнорируем HATEOAS и сосредоточимся на простых API без поддержки гипермедиа.

Имитируя работу клиента этого API, мы можем использовать curl, чтобы перейти по ссылке `self` и получить сведения об ингредиенте «Flour Tortilla» (мучная лепешка):

```
$ curl http://localhost:8080/ingredients/FLTO
{
  "name" : "Flour Tortilla",
  "type" : "WRAP",
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/ingredients/FLTO"
    },
    "ingredient" : {
      "href" : "http://localhost:8080/ingredients/FLTO"
    }
  }
}
```

Чтобы не слишком отвлекаться, мы не будем тратить много времени на изучение каждой конечной точки, созданной Spring Data REST. Но вы должны знать, что они также поддерживают методы POST, PUT и DELETE. Все верно: вы можете отправить запрос POST в конечную точку `/ingredients`, чтобы создать новый ингредиент, и DELETE в `/ingredients/FLTO`, чтобы удалить мучные лепешки из меню.

Одна из настроек, которую вы, возможно, захотите выполнить, – определить базовый путь для API, чтобы его конечные точки не конфликтовали с вашими контроллерами. Чтобы настроить базовый путь для API, нужно установить свойство `spring.data.rest.base-path`:

```
spring:
  data:
    rest:
      base-path: /data-api
```

Эта конфигурация устанавливает базовый путь `/data-api` для конечных точек Spring Data REST. Вы можете установить базовый путь, какой только пожелаете, и в данном случае выбор `/data-api` гарантирует, что конечные точки, предоставляемые Spring Data REST, не будут конфликтовать ни с какими другими контроллерами, включая созданные нами выше в этой главе, чей путь начинается с `/api`. Как следствие для доступа к ингредиентам теперь будет использоваться конечная точка `/data-api/ingredients`. Попробуйте этот новый базовый путь, запросив список рецептов тако:

```
$ curl http://localhost:8080/data-api/tacos
{
  "timestamp": "2018-02-11T16:22:12.381+0000",
  "status": 404,
  "error": "Not Found",
  "message": "No message available",
  "path": "/api/tacos"
}
```

Что такое? Почему ничего не получилось? У нас есть сущность `Ingredient` и интерфейс `IngredientRepository`, доступ к которому Spring Data REST предоставляет посредством конечной точки `/data-api/ingredients`. У нас также есть объект `Taco` и интерфейс `TacoRepository`, но почему Spring Data REST не предоставляет конечную точку `/data-api/tacos`?

7.2.1 Настройка имен отношений и путей к ресурсам

На самом деле Spring Data REST предоставляет конечную точку для работы с рецептами тако. Но какой бы умной ни была библиотека Spring Data REST, она иногда путается в выборе имен конечных точек, что мы и видели в примере выше.

При создании конечных точек для репозиториев Spring Data библиотека Spring Data REST пытается использовать имена классов сущностей во множественном числе. Для объекта `Ingredient` конечной точкой является `/data-api/ingredients`. Для объекта `TacoOrder` – `/data-api/orders`.

Но иногда, как, например, в случае с «`taco`», она спотыкается и генерирует не совсем верный вариант во множественном числе. Как оказывается, Spring Data REST преобразует слово «`taco`» во множественное число как «`tacoes`», поэтому, чтобы запросить тако, нужно подыграть и обратиться к конечной точке `/data-api/tacoes`:

```
$ curl localhost:8080/data-api/tacoes
{
```

```
"_embedded" : {
    "tacoes" : [ {
        "name" : "Carnivore",
        "createdAt" : "2018-02-11T17:01:32.999+0000",
        "_links" : {
            "self" : {
                "href" : "http://localhost:8080/data-api/tacoes/2"
            },
            "taco" : {
                "href" : "http://localhost:8080/data-api/tacoes/2"
            },
            "ingredients" : {
                "href" : "http://localhost:8080/data-api/tacoes/2/ingredients"
            }
        }
    }]
},
"page" : {
    "size" : 20,
    "totalElements" : 3,
    "totalPages" : 1,
    "number" : 0
}
}
```

Возможно, вам интересно узнать, как я определил, что слово «taco» ошибочно преобразуется в «tacoes». Как оказалось, Spring Data REST также предоставляет домашний ресурс, в котором перечислены ссылки на все имеющиеся конечные точки. Чтобы получить этот список, достаточно выполнить запрос GET к базовому пути API:

```
$ curl localhost:8080/api
{
    "_links" : {
        "orders" : {
            "href" : "http://localhost:8080/data-api/orders"
        },
        "ingredients" : {
            "href" : "http://localhost:8080/data-api/ingredients"
        },
        "tacoes" : {
            "href" : "http://localhost:8080/data-api/tacoes{?page,size,sort}",
            "templated" : true
        },
        "users" : {
            "href" : "http://localhost:8080/data-api/users"
        },
        "profile" : {
            "href" : "http://localhost:8080/data-api/profile"
        }
    }
}
```

Как видите, домашний ресурс возвращает ссылки, соответствующие всем имеющимся сущностям. Все ссылки выглядят верными, кроме `tacos`, где имя отношения и URL имеют неверное написание слова «`taco`» во множественном числе.

Однако все не так плохо, и вы не обязаны мириться с этой маленькой особенностью Spring Data REST. Добавив следующую простую аннотацию к классу `Taco`, можно настроить имя отношения и сам путь:

```
@Data
@Entity
@RestResource(rel="tacos", path="tacos")
public class Taco {
    ...
}
```

Аннотация `@RestResource` позволяет дать сущности любое имя отношения и путь. В данном случае мы задали имя и путь `"tacos"`. Если теперь выполнить запрос к домашнему ресурсу, в ответ будетозвращена ссылка `tacos` с правильным написанием во множественном числе:

```
"tacos" : {
    "href" : "http://localhost:8080/data-api/tacos[?page,size,sort}",
    "templated" : true
},
```

Теперь мы можем использовать конечную точку `/data-api/tacos` для работы с рецептами тако.

А сейчас давайте посмотрим, как можно упорядочивать результаты, получаемые из конечных точек Spring Data REST.

7.2.2 Деление на страницы и упорядочение

Возможно, вы заметили, что все ссылки из домашнего ресурса поддерживают дополнительные параметры `page`, `size` и `sort`. По умолчанию запросы к ресурсам коллекций, таких как `/data-api/tacos`, будут возвращать их содержимое страницами по 20 элементов на каждой, начиная с первой. Однако есть возможность настроить размер страницы и указать номер возвращаемой страницы, передав параметры `page`, `size` и `sort` в запросе.

Например, чтобы запросить первую страницу с рецептами тако, где размер страницы равен 5, можно отправить такой запрос GET (в этом примере используется утилита `curl`):

```
$ curl "localhost:8080/data-api/tacos?size=5"
```

Предполагая, что будет видно более пяти рецептов тако, можно запросить вторую страницу, добавив параметр `page`:

```
$ curl "localhost:8080/data-api/tacos?size=5&page=1"
```

Обратите внимание, что нумерация страниц начинается с нуля, то есть запрос с параметром `page=1` фактически запрашивает вторую страницу. (Отметьте также, что многие командные оболочки по-своему интерпретируют амперсанд в запросе, поэтому в предыдущем запросе я заключил URL в кавычки.)

Параметр `sort` позволяет отсортировать список по любому свойству сущности. Например, допустим, что нам нужно получить 12 последних созданных рецептов тако для отображения в пользовательском интерфейсе. Это можно сделать, указав следующее сочетание параметров `page` и `sort`:

```
$ curl "localhost:8080/data-api/tacos?sort=createdAt,desc&page=0&size=12"
```

Здесь параметр `sort` указывает, что рецепты должны упорядочиваться по свойству `createdDate` в порядке убывания (чтобы самые новые рецепты стояли в списке первыми). Параметры `page` и `size` сообщают, что первая страница должна содержать 12 рецептов тако.

Это именно то, что нужно пользовательскому интерфейсу для отображения последних созданных рецептов тако. Примерно так действует и конечная точка `/api/tacos?recent`, которую мы определили в `TacoController` выше в этой главе.

Теперь переключим наше внимание и посмотрим, как на стороне клиента использовать созданные нами конечные точки API.

7.3 Использование служб REST

Случалось ли с вами, когда вы приходили в кино и обнаруживали, что к началу фильма оставались единственным зрителем в зале? Это, безусловно, интересный опыт фактически частного просмотра фильма. Вы можете выбрать любое место, поговорить с персонажами на экране и даже открыть свой телефон и написать об этом в Твиттере, и никто не рассердится за то, что вы помешали просмотру фильма. Но самое приятное, что никто другой не помешает вам!

Со мной такое случалось нечасто, но когда это происходило, я задавался вопросом, что случилось бы, если бы я не появился. Показали бы фильм? Спас бы герой мир? Проводили бы уборку сотрудники кинотеатра после окончания фильма?

Фильм без зрителей – это как API без клиента. Он готов принимать запросы и возвращать данные, но если никто не обращается к API, то можно ли назвать его API? API чем-то похож на кота Шредингера – мы не можем знать, активен ли API и возвращает ли он ответы HTTP 404, пока не отправим ему запрос.

Приложения Spring нередко предоставляют API и посылают запросы к API других приложений. Это особенно верно в мире микросервисов. Поэтому стоит уделить немного времени изучению особенностей Spring для взаимодействия с REST API.

Приложение Spring может использовать REST API с помощью следующих инструментов:

- *RestTemplate* – простой синхронный клиент REST, предоставляемый ядром Spring Framework;
- *Traverser* – обертка вокруг *RestTemplate*, предоставляемая механизмом Spring HATEOAS, позволяющая использовать синхронного клиента REST с поддержкой гиперссылок и созданная по образу и подобию одноименной библиотеки JavaScript;
- *WebClient* – реактивный асинхронный клиент REST.

В этой главе мы сосредоточимся на создании клиентов с помощью *RestTemplate*. А обсуждение *WebClient* отложим до главы 1, где рассматривается реактивная веб-инфраструктура Spring. А если вам интересно узнать, как создавать клиентов с поддержкой гиперссылок, то обращайтесь к документации *Traverser*, доступной по адресу <http://mng.bz/aZno>.

Взаимодействие с ресурсом REST требует написать довольно много кода на стороне клиента, в основном шаблонного. Клиент, использующий низкоуровневые библиотеки HTTP, должен создать экземпляр клиента и объект запроса, выполнить запрос, интерпретировать ответ, преобразовать ответ в объекты предметной области и обработать любые исключения, которые могут возникнуть. И весь этот шаблонный код приходится повторять снова и снова, независимо типа отправляемого HTTP-запроса.

Чтобы избежать необходимости писать такой шаблонный код, Spring предоставляет *RestTemplate*. Подобно *JdbcTemplate*, который избавляет от утомительных тонкостей при работе с JDBC, *RestTemplate* освобождает от необходимости подробно описывать все детали использования ресурсов REST.

RestTemplate предлагает 41 метод для взаимодействия с ресурсами REST. Но мы рассмотрим только дюжину уникальных операций, каждая из которых имеет перегруженные версии, составляющие полный набор из 41 метода. Эти 12 операций описаны в табл. 7.2.

За исключением TRACE, класс *RestTemplate* имеет по крайней мере один метод для каждого из стандартных методов HTTP. Кроме того, в лице `execute()` и `exchange()` предоставляются низкоуровневые универсальные методы для отправки любых видов запросов HTTP.

Большинство методов из табл. 7.2 имеют три перегруженные версии:

- версия, принимающая строку URL с параметрами, указанными в дополнительных аргументах;
- версия, принимающая строку URL с параметрами в форме `Map<String, String>`;
- версия, принимающая URL в форме `java.net.URI` без поддержки параметризованных URL.

Познакомившись с 12 операциями, предоставляемыми классом *RestTemplate*, и особенностями их использования, вы сможете приступить к разработке своих REST-клиентов.

Таблица 7.2 RestTemplate определяет 12 уникальных операций, каждая из которых имеет перегруженные версии, составляющие полный набор из 41 метода

Метод	Описание
<code>delete(...)</code>	Выполняет HTTP-запрос DELETE к ресурсу с указанным URL
<code>exchange(...)</code>	Выполняет указанный HTTP-запрос с заданным URL; возвращает <code>ResponseEntity</code> с объектом, соответствующим телу ответа
<code>execute(...)</code>	Выполняет указанный HTTP-запрос с заданным URL; возвращает объект, соответствующий телу ответа
<code>getForEntity(...)</code>	Выполняет HTTP-запрос GET; возвращает <code>ResponseEntity</code> с объектом, соответствующим телу ответа
<code>getForObject(...)</code>	Выполняет HTTP-запрос GET; возвращает объект, соответствующий телу ответа
<code>headForHeaders(...)</code>	Выполняет HTTP-запрос HEAD; возвращает HTTP-заголовки для указанного URL ресурса
<code>optionsForAllow(...)</code>	Выполняет HTTP-запрос OPTIONS; возвращает заголовок <code>Allow</code> для указанного URL
<code>patchForObject(...)</code>	Выполняет HTTP-запрос PATCH; возвращает объект, соответствующий телу ответа
<code>postForEntity(...)</code>	Выполняет HTTP-запрос POST, отправляя данные по адресу URL; возвращает <code>ResponseEntity</code> с объектом, соответствующим телу ответа
<code>postForLocation(...)</code>	Выполняет HTTP-запрос POST, отправляя данные по адресу URL; возвращает вновь созданного ресурса
<code>postForObject(...)</code>	Выполняет HTTP-запрос POST, отправляя данные по адресу URL; возвращает объект, соответствующий телу ответа
<code>put(...)</code>	Выполняет HTTP-запрос PUT, отправляя данные по адресу URL

Чтобы использовать `RestTemplate`, нужно создать экземпляр этого класса, как показано ниже:

```
RestTemplate rest = new RestTemplate();
```

или объявить его bean-компонентом и внедрить туда, где он необходим:

```
@Bean
public RestTemplate restTemplate() {
    return new RestTemplate();
}
```

Давайте рассмотрим операции `RestTemplate`, поддерживающие четыре основных метода HTTP: GET, PUT, DELETE и POST. Начнем с `getForObject()` и `getForEntity()`, выполняющих HTTP-запросы GET.

7.3.1 Получение ресурса запросом GET

Предположим, что нам нужно получить ингредиент из Taco Cloud API. Сделать это можно с помощью метода `getForObject()` класса `RestTemplate`. Например, следующий код получает объект `Ingredient` по его идентификатору:

```
public Ingredient getIngredientById(String ingredientId) {
    return rest.getForObject("http://localhost:8080/ingredients/{id}",
        Ingredient.class, ingredientId);
}
```

Здесь мы используем версию `getForObject()`, которая принимает строку с URL и список аргументов для подстановки в параметры URL. Аргумент `ingredientId`, переданный в `getForObject()`, подставляется на место `{id}` в URL. В этом примере используется только один параметр URL, но вообще подстановка аргументов в параметры производится в том порядке, в котором они заданы в вызове метода.

Второй параметр `getForObject()` – это тип объекта, в котором должен быть возвращен ответ. В данном случае ответ (скорее всего, в формате JSON) должен быть десериализован в объект `Ingredient`.

Также для передачи параметров URL можно использовать ассоциативный массив, как показано ниже:

```
public Ingredient getIngredientById(String ingredientId) {
    Map<String, String> urlVariables = new HashMap<>();
    urlVariables.put("id", ingredientId);
    return rest.getForObject("http://localhost:8080/ingredients/{id}",
        Ingredient.class, urlVariables);
}
```

В этом примере аргумент `ingredientId` сопоставляется с ключом `id`. Перед выполнением запроса на место `{id}` подставляется значение из ассоциативного массива, соответствующее ключу `id`.

Использовать параметр URI немного сложнее, так как для этого нужно создать объект URI перед вызовом `getForObject()`. В остальном выполнение запроса мало отличается от двух других способов:

```
public Ingredient getIngredientById(String ingredientId) {
    Map<String, String> urlVariables = new HashMap<>();
    urlVariables.put("id", ingredientId);
    URI url = UriComponentsBuilder
        .fromHttpUrl("http://localhost:8080/ingredients/{id}")
        .build(urlVariables);
    return rest.getForObject(url, Ingredient.class);
}
```

Здесь объект URI создается на основе строкового значения, заполнители в котором замещаются значениями из ассоциативного массива Map, как в предыдущем варианте использования `getForObject()`. Метод `getForObject()` – это довольно мощный инструмент для получения ресурсов. Но если клиенту недостаточно тела ответа, то можно прибегнуть к методу `getForEntity()`.

`getForEntity()` действует почти так же, как `getForObject()`, но вместе с объектом данных предметной области, содержащего информацию из тела ответа, возвращает объект `ResponseEntity` – обертку, включающую объект данных предметной области. Объект `ResponseEntity` содержит дополнительные сведения об ответе, например заголовки ответа.

Предположим, что кроме фактических данных об ингредиентах нам нужно также проверить заголовок `Date` ответа. Вот как это можно сделать с помощью `getForEntity()`:

```
public Ingredient getIngredientById(String ingredientId) {
    ResponseEntity<Ingredient> responseEntity =
        rest.getForEntity("http://localhost:8080/ingredients/{id}",
                          Ingredient.class, ingredientId);
    log.info("Fetched time: {}", responseEntity.getHeaders().getDate());
    return responseEntity.getBody();
}
```

Метод `getForEntity()` имеет такие же перегруженные версии, что и `getForObject()`, поэтому вы можете передавать переменные для включения в URL в качестве параметров в виде списка аргументов или вызвать `getForEntity()` с объектом URI.

7.3.2 Отправка ресурса запросом PUT

Для отправки HTTP-запросов PUT класс `RestTemplate` предлагает метод `put()`. Все три перегруженные версии `put()` принимают объект, который требуется сериализовать и отправить по указанному URL. Что касается самого URL, его можно передать в виде объекта `URI` или строки. Так же, как в случае с `getForObject()` и `getForEntity()`, значения для параметров URL могут передаваться в виде списка аргументов или ассоциативного массива.

Предположим, нам нужно заменить ресурс ингредиента данными из нового объекта `Ingredient`. Вот как это можно сделать:

```
public void updateIngredient(Ingredient ingredient) {
    rest.put("http://localhost:8080/ingredients/{id}",
             ingredient, ingredient.getId());
}
```

Здесь URL передается в виде строки и включает параметр, который заменяется свойством `id` объекта `Ingredient`. Отправляемые данные – это сам объект `Ingredient`. Метод `put()` ничего не возвращает, поэтому нам не нужно ничего предпринимать для обработки возвращаемого значения.

7.3.3 Удаление ресурса запросом DELETE

Предположим, что в Taco Cloud решили исключить из меню некоторый ингредиент и теперь его нужно сделать недоступным для использования в рецептах. Сделать это можно вызовом метода `delete()` класса `RestTemplate`:

```
public void deleteIngredient(Ingredient ingredient) {
    rest.delete("http://localhost:8080/ingredients/{id}",
                ingredient.getId());
}
```

В этом примере в вызов метода `delete()` передаются только URL (в виде строки) и значение параметра URL. Так же как в случае с другими

методами `RestTemplate`, URL можно передать в виде объекта `URI` или в виде строки и со значениями параметров в ассоциативном массиве.

7.3.4 Отправка ресурса запросом POST

Теперь предположим, что в Taco Cloud решили добавить в меню новый ингредиент. Сделать его доступным для составления рецептов можно с помощью HTTP-запроса `POST` к конечной точке `.../ingredients` и с данными об ингредиенте в теле запроса. Класс `RestTemplate` предлагает три метода отправки `POST`-запроса, каждый из которых имеет одинаковые перегруженные версии, отличающиеся способом передачи URL. Добавить новый ресурс `Ingredient` с помощью запроса `POST` можно вызовом `postForObject()`:

```
public Ingredient createIngredient(Ingredient ingredient) {
    return rest.postForObject("http://localhost:8080/ingredients",
                               ingredient, Ingredient.class);
}
```

Эта версия метода `postForObject()` принимает строку с URL, объект для отправки на сервер и тип данных предметной области для возврата тела ответа. В четвертом необязательном параметре можно передать ассоциативный массив `Map` со значениями параметров для подстановки в URL, но в данном примере мы не используем эту возможность.

Если после отправки `POST`-запроса нужно получить местоположение только что созданного ресурса, то запрос можно выполнить вызовом `postForLocation()`:

```
public java.net.URI createIngredient(Ingredient ingredient) {
    return rest.postForLocation("http://localhost:8080/ingredients",
                                ingredient);
}
```

Обратите внимание, что `postForLocation()` работает почти так же, как `postForObject()`, но вместо объекта ресурса возвращает `URI` со ссылкой на только что созданный ресурс. Возвращаемый экземпляр `URI` создается на основе заголовка `Location` ответа. Если вам понадобятся и сам объект, и его адрес, то используйте `postForEntity()`:

```
public Ingredient createIngredient(Ingredient ingredient) {
    ResponseEntity<Ingredient> responseEntity =
        rest.postForEntity("http://localhost:8080/ingredients",
                           ingredient,
                           Ingredient.class);
    log.info("New resource created at {}",
            responseEntity.getHeaders().getLocation());
    return responseEntity.getBody();
}
```

Несмотря на различия в предназначении, методы `RestTemplate` очень похожи особенностями использования. Это упрощает осваивание `RestTemplate` и его практическое использование.

Итоги

- Конечные точки REST можно создавать с помощью Spring MVC, используя ту же модель программирования контроллеров, как и при обслуживании браузеров.
- Методы-обработчики контроллера могут снабжаться аннотацией `@ResponseBody` или возвращать объекты `ResponseEntity`, чтобы записывать данные непосредственно в тело ответа, минуя модели данных и представления.
- Аннотация `@RestController` упрощает разработку контроллеров REST, устранивая необходимость использовать `@ResponseBody` в методах-обработчиках.
- Репозитории Spring Data могут автоматически генерировать конечные точки REST API с помощью Spring Data REST.

Безопасность REST API



В этой главе рассматриваются следующие темы:

- защита API с помощью OAuth 2;
- создание сервера авторизации;
- добавление сервера ресурсов в API;
- использование API, защищенного с помощью OAuth 2.

Приходилось ли вам когда-нибудь пользоваться услугами парковщика? Суть проста: вы передаете ключи от машины служащему у входа в магазин, гостиницу, театр или ресторан, а он заботится о том, чтобы найти место на парковке и поставить туда вашу машину. А потом, когда вы соберетесь покинуть заведение, вам вернут вашу машину. Возможно, потому, что я много раз смотрел фильм «Выходной день Ферриса Бьюллера», но мне никогда не хотелось передавать ключи от моей машины незнакомцу в надежде, что он позаботится о моей машине вместо меня.

Услуга парковщика предполагает предоставление постороннему лицу права заботиться о вашем автомобиле. Во многих новых автомобилях предусмотрен «ключ парковщика», специальный ключ, с помощью которого можно только открыть двери автомобиля и запустить двигатель. То есть объем доверия ограничен. Парковщик не сможет открыть бардачок или багажник ключом парковщика.

В распределенных приложениях доверие между программными системами имеет решающее значение. Даже в простой ситуации, ког-

да клиентское приложение использует серверный API, важно, чтобы конкретный клиент был доверенным, а все остальные, пытающиеся использовать тот же API, блокировались. По аналогии с парковщиком, степень доверия, оказываемого клиенту, должна быть ограничена только функциями, необходимыми ему для выполнения своей работы.

Защита REST API отличается от защиты браузерного веб-приложения. В этой главе мы рассмотрим OAuth 2 – спецификацию авторизации, созданную специально для обеспечения безопасности API. При этом основное внимание мы будем уделять поддержке OAuth 2 в Spring Security. Но сначала подготовим почву и посмотрим, как работает OAuth 2.

8.1 Знакомство с OAuth 2

Предположим, что нам нужно создать новый инструмент для управления приложением Taco Cloud и, в частности, для управления списком ингредиентов, доступных на основном веб-сайте Taco Cloud.

Прежде чем начать писать код для этого инструмента, добавим несколько новых конечных точек в Taco Cloud API для поддержки управления ингредиентами. Контроллер REST в листинге 8.1 реализует три конечные точки для получения списка, добавления и удаления ингредиентов.

Листинг 8.1 Контроллер для управления доступными ингредиентами

```
package tacos.web.api;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.DeleteMapping;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseStatus;
import org.springframework.web.bind.annotation.RestController;

import tacos.Ingredient;
import tacos.data.IngredientRepository;
@RestController
@RequestMapping(path="/api/ingredients", produces="application/json")
@CrossOrigin(origins="http://localhost:8080")
public class IngredientController {

    private IngredientRepository repo;
```

```

@.Autowired
public IngredientController(IngredientRepository repo) {
    this.repo = repo;
}

@GetMapping
public Iterable<Ingredient> allIngredients() {
    return repo.findAll();
}

@PostMapping
@ResponseBody(HttpStatus.CREATED)
public Ingredient saveIngredient(@RequestBody Ingredient ingredient) {
    return repo.save(ingredient);
}

@DeleteMapping("/{id}")
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteIngredient(@PathVariable("id") String ingredientId) {
    repo.deleteById(ingredientId);
}
}

```

Отлично! Теперь можно приступать к работе над инструментом администрирования, который будет вызывать эти конечные точки по мере необходимости для добавления и удаления ингредиентов.

Но подождите – этот API пока не имеет никакой защиты. То есть не только наш инструмент, но и любое другое приложение сможет отправлять HTTP-запросы, чтобы добавить или удалить ингредиенты. Даже с помощью клиента командной строки curl любой посторонний сможет добавить новый ингредиент:

```
$ curl localhost:8080/ingredients \
-H "Content-type: application/json" \
-d'{"id":"FISH", "name":"Stinky Fish", "type":"PROTEIN"}'
```

или удалить существующий¹:

```
$ curl localhost:8080/ingredients/GRBF -X DELETE
```

Этот API является частью основного приложения и доступен всему миру; на самом деле конечная точка GET используется пользовательским интерфейсом основного приложения в *home.html*. Очевидно, что нам нужно защитить, по меньшей мере, конечные точки POST и DELETE.

¹ В зависимости от используемой базы данных и схемы, если ингредиент уже является частью существующего рецепта тако, настройки ограничения целостности могут предотвратить удаление. Но кроме схемы базы данных нет никаких других ограничений, которые могли бы помешать удалить ингредиент.

Один из вариантов – использовать для защиты конечных точек */ingredients* аутентификацию HTTP Basic. Это можно сделать, добавив аннотацию `@PreAuthorize` к методам-обработчикам:

```
@PostMapping
@PreAuthorize("#{hasRole('ADMIN')}")
public Ingredient saveIngredient(@RequestBody Ingredient ingredient) {
    return repo.save(ingredient);
}

@DeleteMapping("/{id}")
@PreAuthorize("#{hasRole('ADMIN')}")
public void deleteIngredient(@PathVariable("id") String ingredientId) {
    repo.deleteById(ingredientId);
}
```

или определив настройки в конфигурации безопасности:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        .antMatchers(HttpMethod.POST, "/ingredients").hasRole("ADMIN")
        .antMatchers(HttpMethod.DELETE, "/ingredients/**").hasRole("ADMIN")

        ...
}
```

Когда использовать или не использовать префикс «ROLE_»

В Spring Security поддерживается несколько форм привилегий, включая роли, разрешения и (как мы увидим позже) области действия OAuth2. Роли, в частности, – это особая форма привилегий, названия которых начинаются с префикса "ROLE_".

При использовании методов или выражений SpEL, имеющих дело с ролями, таких как `hasRole()`, префикс "ROLE_" подразумевается по умолчанию. Поэтому вызов `hasRole("ADMIN")` фактически выполнит проверку привилегий с именем "ROLE_ADMIN". При вызове таких методов и функций не нужно использовать префикс "ROLE_" (иначе это приведет к дублированию префикса "ROLE_").

Другие методы и функции в Spring Security, не зависящие от конкретного типа полномочий, тоже можно использовать для проверки ролей, но в подобных случаях необходимо явно добавить префикс "ROLE_". Например, если вы решите использовать `hasAuthority()` вместо `hasRole()`, то вам нужно будет передать в вызов "ROLE_ADMIN" вместо "ADMIN".

В любом случае, возможность отправлять запросы POST или DELETE конечной точке */ingredients* потребует, чтобы отправитель также ввел учетные, наделенные привилегиями "ROLE_ADMIN". При использова-

нии curl, например, учетные данные можно указать с помощью параметра -u:

```
$ curl localhost:8080/ingredients \
-H "Content-type: application/json" \
-d'{"id":"FISH","name":"Stinky Fish", "type":"PROTEIN"}' \
-u admin:l3tm31n
```

HTTP Basic, конечно, защитит API, но эта защита... гм... уж слишком простенькая. Она требует, чтобы клиент и API использовали общие учетные данные пользователя. Более того, HTTP Basic шифрует учетные данные в заголовке запроса с помощью алгоритма Base64, и если хакер каким-то образом перехватит запрос, он с легкостью расшифрует учетные данные и сможет использовать их во вред. Если такое произойдет, то придется изменить пароль и обновить и повторно аутентифицировать всех клиентов.

А что, если вместо аутентификации пользователя-администратора при каждом запросе API будет просто запрашивать некоторый токен, доказывающий, что пользователь обладает необходимыми привилегиями? Это примерно как билет на спортивное мероприятие. Чтобы пройти на трибуны, контролеру у турнекта не нужно знать, кто вы: если вы предъявите действительный билет, то он пропустит вас.

Примерно так работает авторизация OAuth 2. Клиент запрашивает токен доступа – аналог ключа парковщика – на сервере авторизации с явного разрешения пользователя. Этот токен позволяет взаимодействовать с API от имени пользователя, авторизовавшего клиента. В любой момент срок действия токена может истечь или он может быть отозван без смены пароля пользователем. В таких случаях клиент просто запросит новый токен доступа, чтобы иметь возможность продолжать действовать от имени пользователя. Этот процесс показан на рис. 8.1.

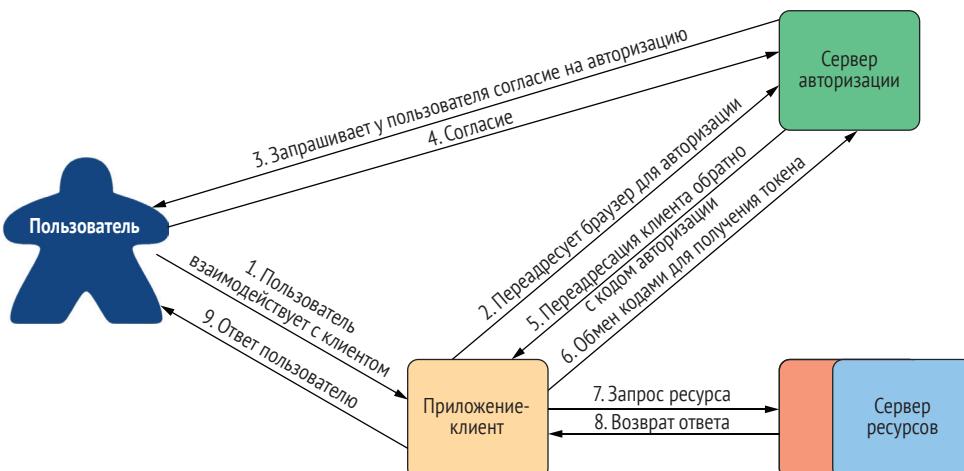


Рис. 8.1 Процедура авторизации с помощью OAuth 2

OAuth 2 – очень богатая спецификация безопасности, поддерживающая множество способов ее использования. Процесс, изображенный на рис. 8.1, называется *предоставлением кода авторизации*. В числе других процессов, поддерживаемых спецификацией OAuth 2, можно назвать:

- *неявное предоставление разрешений* – по аналогии с предоставлением кода авторизации, неявное предоставление разрешений переадресует браузер на сервер авторизации для получения согласия пользователя. Но при обратной переадресации вместо кода авторизации в запрос неявно подставляется токен. Изначально этот процесс разрабатывался для клиентов JavaScript, работающих в браузере, и в настоящее время не рекомендуется к использованию; вместо него следует использовать процесс предоставления кода авторизации;
- *предоставление учетных данных (пароля) пользователя* – этот процесс не предусматривает переадресации и вообще может не использовать веб-браузер. Вместо этого клиентское приложение получает учетные данные непосредственно от пользователя и обменивает их на токен доступа. Этот процесс выглядит подходящим для клиентов, не основанных на браузере, однако многие современные приложения предпочитают просить пользователя перейти на веб-сайт в своем браузере и пройти процесс предоставления кода авторизации, чтобы избежать необходимости получать и обрабатывать учетные данные пользователя;
- *предоставление учетных данных клиента* – этот процесс похож на предоставление учетных данных пользователя, только вместо учетных данных пользователя клиент посыпает серверу авторизации свои собственные учетные данные. Однако токен, предоставляемый в ответ, позволяет выполнить лишь ограниченный круг операций, не связанных с пользователем, и не может применяться для выполнения действий от имени пользователя.

Далее мы будем использовать процесс предоставления кода авторизации, чтобы получить токен доступа JSON Web Token (JWT). Для этого нам потребуется создать несколько приложений, взаимодействующих друг с другом:

- *сервер авторизации* – задача сервера авторизации заключается в том, чтобы получить от пользователя разрешение, позволяющее клиентскому приложению выступать от его имени. Если пользователь даст разрешение, то сервер авторизации предоставит токен доступа клиентскому приложению, который оно сможет использовать для доступа к API;
- *сервер ресурсов* – это просто другое название API, защищенного с помощью OAuth 2. Сервер ресурсов является частью самого API, однако в дальнейшем обсуждении мы будем разделять эти два понятия. Сервер ресурсов ограничивает доступ к своим ресурсам, если запрос не предоставляет доступ к действительному токену с необходимым набором разрешений. Для нас сервером

ресурсов будет служить Taco Cloud API, который мы начали создавать в главе 7, мы лишь добавим в него некоторые настройки безопасности;

- *клиентское приложение* – это приложение, использующее API, для доступа к которому требуется разрешение. Мы напишем простое приложение администратора для Taco Cloud, которое позволит добавлять новые ингредиенты;
- *пользователь* – человек, использующий клиентское приложение и дающий ему разрешение на доступ к API сервера ресурсов от своего имени.

В процессе предоставления кода авторизации для получения токена доступа выполняется последовательность переадресаций браузера между клиентским приложением и сервером авторизации. Все начинается с того, что клиент переадресует браузер пользователя на сервер авторизации, запрашивая определенные разрешения (или «область действия»). Затем сервер авторизации просит пользователя выполнить вход и дать согласие на предоставление запрошенных разрешений. Когда пользователь даст согласие, сервер авторизации переадресует браузер обратно в клиентское приложение с кодом, который затем клиент может обменять на токен доступа. Получив токен, клиент может использовать его для взаимодействия с API сервера ресурсов, передавая в заголовке "Authorization" с каждым запросом.

В этом разделе мы ограничимся исследованием конкретного примера использованием OAuth 2, но вообще я советую изучить тему авторизации глубже, прочитав спецификацию OAuth 2 (<https://oauth.net/2/>) или любую из следующих книг по этой теме:

- *OAuth 2 in Action*: <https://www.manning.com/books/oauth-2-in-action>;
- *Microservices Security in Action*: <https://www.manning.com/books/microservices-security-in-action>;
- *API Security in Action*: <https://www.manning.com/books/api-security-in-action>.

Также можете заглянуть в проект под названием «Protecting User Data with Spring Security and OAuth2» (<http://mng.bz/4KdD>).

В течение нескольких лет проект под названием Spring Security for OAuth поддерживал обе версии – OAuth 1.0a и OAuth 2. Он не входил в состав фреймворка Spring Security, но разрабатывался той же командой. Однако несколько лет назад команда Spring Security включила компоненты клиента и сервера ресурсов в фреймворк.

Но сервер авторизации решили не включать в Spring Security. Разработчикам рекомендуется использовать существующие серверы авторизации от сторонних поставщиков, таких как Okta, Google и др. Учитывая высокий спрос в сообществе, команда Spring Security запустила проект Spring Authorization Server¹. Этот проект отмечен как

¹ <http://mng.bz/QqGR>.

«экспериментальный» и должен продвигаться сообществом, но все же он служит отличным способом начать работу с OAuth 2 без регистрации любой из сторонних реализаций сервера авторизации.

В оставшейся части данной главы мы посмотрим, как использовать механизм OAuth 2 с помощью Spring Security. Попутно создадим два новых проекта, проект сервера авторизации и проект клиента, и изменим существующий проект Taco Cloud, преобразовав его API в сервер ресурсов. Начнем с создания сервера авторизации с помощью проекта Spring Authorization Server.

8.2 Создание сервера авторизации

Основная задача сервера авторизации – выдать токен доступа, разрешающий выполнять действия от имени пользователя. Как упоминалось выше, у нас на выбор есть несколько реализаций сервера авторизации, но мы будем использовать Spring Authorization Server в нашем проекте. Spring Authorization Server является экспериментальным и реализует не все процессы OAuth 2, однако он поддерживает представление кода авторизации и учетных данных клиента.

Сервер авторизации – это приложение, отличное от приложения, реализующего API, а также от клиента. Поэтому, чтобы начать работу с Spring Authorization Server, нужно создать новый проект Spring Boot, выбрав (по крайней мере) начальные зависимости для поддержки веба и безопасности. Информацию о пользователях наш сервер авторизации будет хранить в реляционной базе данных с использованием JPA, поэтому добавим также начальные зависимости JPA и H2. А если вы предпочитаете использовать библиотеку Lombok для автоматического создания методов доступа, конструкторов и многоного другого, обязательно включите и ее.

Зависимость Spring Authorization Server недоступна (пока) в Initializr. Поэтому сразу после создания проекта нужно вручную добавить зависимость Spring Authorization Server в спецификацию сборки. Для примера ниже приводится зависимость Maven, которую нужно включить в файл *pom.xml*:

```
<dependency>
    <groupId>org.springframework.security.experimental</groupId>
    <artifactId>spring-security-oauth2-authorization-server</artifactId>
    <version>0.1.2</version>
</dependency>
```

Далее, поскольку все приложения будут запускаться на машине для разработки (по крайней мере, пока), нужно убедиться, что между основным приложением Taco Cloud и сервером авторизации нет конфликта по портам. Следующая запись в файле проекта *application.yml* сделает сервер авторизации доступным на порту 9000:

```
server:
  port: 9000
```

Теперь углубимся в базовую конфигурацию безопасности сервера авторизации. В листинге 8.2 показан очень простой класс конфигурации Spring Security, реализующий форму входа для аутентификации всех запросов.

Листинг 8.2 Конфигурация безопасности, основанная на форме ввода учетных данных

```
package tacos.authorization;
import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.web.builders.
    HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.
    EnableWebSecurity;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;

import tacos.authorization.users.UserRepository;

@EnableWebSecurity
public class SecurityConfig {

    @Bean
    SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http)
        throws Exception {
        return http
            .authorizeRequests(authorizeRequests ->
                authorizeRequests.anyRequest().authenticated()
            )
            .formLogin()
            .and().build();
    }

    @Bean
    UserDetailsService userDetailsService(UserRepository userRepo) {
        return username -> userRepo.findByUsername(username);
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

Обратите внимание, что bean-компонент `UserDetailsService` использует `TacoUserRepository` для поиска пользователей по их именам. Но нас сейчас интересует конфигурация самого сервера авторизации,

а не репозиторий TacoUserRepository, поэтому отмечу лишь, что этот репозиторий очень похож на другие репозитории Spring Data, созданные нами начиная с главы 3.

Единственное, о чем следует упомянуть особо, – TacoUserRepository можно использовать в bean-компоненте CommandLineRunner для предварительного заполнения базы данных парой тестовых пользователей:

```
@Bean
public ApplicationRunner dataLoader(
    UserRepository repo, PasswordEncoder encoder) {
    return args -> {
        repo.save(
            new User("habuma", encoder.encode("password"), "ROLE_ADMIN"));
        repo.save(
            new User("tacochef", encoder.encode("password"), "ROLE_ADMIN"));
    };
}
```

Теперь применим созданную конфигурацию для настройки сервера авторизации. Первым шагом создадим новый класс конфигурации и импортируем общую конфигурацию для настройки сервера авторизации. Следующее определение AuthorizationServerConfig послужит нам хорошим началом:

```
@Configuration(proxyBeanMethods = false)
public class AuthorizationServerConfig {

    @Bean
    @Order(Ordered.HIGHEST_PRECEDENCE)
    public SecurityFilterChain
        authorizationServerSecurityFilterChain(HttpSecurity http) throws
        Exception {
        OAuth2AuthorizationServerConfiguration
            .applyDefaultSecurity(http);
        return http
            .formLogin(Customizer.withDefaults())
            .build();
    }

    ...
}
```

Метод `authenticationServerSecurityFilterChain()` определяет `SecurityFilterChain`, который настраивает некоторое поведение по умолчанию для сервера авторизации OAuth 2 и страницу входа по умолчанию. Аннотация `@Order` присваивает `Ordered.HIGHEST_PRECEDENCE`, гарантируя, что если по какой-то причине будут объявлены другие bean-компоненты этого же типа, то данный компонент будет иметь приоритет над другими.

По большей части это шаблонная конфигурация. Если хотите, то можете изучить ее глубже и скорректировать некоторые аспекты. Но я предлагаю пока просто использовать значения по умолчанию.

Единственный нестандартный компонент, который не предоставляетя OAuth2AuthorizationServerConfiguration, – это репозиторий клиентов. Этот репозиторий подобен службе с информацией о пользователях, только вместо сведений о пользователях он хранит сведения о клиентах, которые могут запрашивать авторизацию от имени пользователей. Он определяется интерфейсом RegisteredClientRepository:

```
public interface RegisteredClientRepository {
    @Nullable
    RegisteredClient findById(String id);

    @Nullable
    RegisteredClient findByClientId(String clientId);

}
```

Для промышленного окружения можно написать свою реализацию RegisteredClientRepository, чтобы получать сведения о клиенте из базы данных или из другого источника. Но по умолчанию Spring Authorization Server использует реализацию в памяти, которая идеально подходит для демонстрации и тестирования. Вы можете реализовать RegisteredClientRepository так, как считете нужным, но в этой книге мы будем использовать реализацию в памяти и зарегистрируем одного клиента на сервере авторизации. Добавим следующий метод в AuthorizationServerConfig:

```
@Bean
public RegisteredClientRepository registeredClientRepository(
    PasswordEncoder passwordEncoder) {
    RegisteredClient registeredClient =
        RegisteredClient.withId(UUID.randomUUID().toString())
            .clientId("taco-admin-client")
            .clientSecret(passwordEncoder.encode("secret"))
            .clientAuthenticationMethod(
                ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)
            .redirectUri(
                "http://127.0.0.1:9090/login/oauth2/code/taco-admin-client")
            .scope("writeIngredients")
            .scope("deleteIngredients")
            .scope(OidcScopes.OPENID)
            .clientSettings(
                clientSettings -> clientSettings.requireUserConsent(true))
            .build();

    return new InMemoryRegisteredClientRepository(registeredClient);
}
```

Как видите, `RegisteredClient` имеет множество деталей. Давайте посмотрим, как определяется наш клиент, двигаясь сверху вниз:

- *ID (withId)* – случайный уникальный идентификатор;
- *идентификатор клиента (clientId)* – аналог имени пользователя, только в роли пользователя выступает клиент. В данном случае "taco-admin-client";
- *секрет клиента (clientSecret)* – аналог пароля для клиента; здесь используется слово "secret";
- *тип авторизации (authorizationGrantType)* – типы разрешений OAuth 2, поддерживаемые клиентом. В данном случае используется код авторизации и токен обновления;
- *URL переадресации (redirectUri)* – один или несколько зарегистрированных URL, куда сервер авторизации может переадресовать клиента после предоставления авторизации. Этот аспект добавляет еще один уровень безопасности, предотвращая получение кода авторизации произвольным приложением, который оно может обменять на токен;
- *область действия (scope)* – одна или несколько областей действия OAuth 2, которые разрешено запрашивать этому клиенту. Здесь используются три области: "writeIngredients", "deleteIngredients" и константа `OidcScopes.OPENID`, которая соответствует области "openid". Область "openid" потребуется позже, когда мы будем использовать сервер авторизации в качестве решения единого входа для приложения администратора Taco Cloud;
- *параметры клиента (clientSettings)* – это лямбда-выражение, позволяющее настраивать параметры клиента. В данном случае мы требуем явного согласия пользователя перед предоставлением доступа к запрошеннной области. Без этого доступ к области предоставлялся бы неявно после входа пользователя.

Наконец, поскольку наш сервер авторизации будет создавать токены JWT, эти токены должны включать подпись, созданную с использованием веб-ключа JSON Web Key (JWK)¹. Поэтому нам понадобится несколько bean-компонентов для создания JWK. Добавим следующий метод (и приватные вспомогательные методы) в `AuthorizationServerConfig`, которые будут решать эту задачу:

```
@Bean
public JWKSource<SecurityContext> jwkSource()
    throws NoSuchAlgorithmException {
    RSAKey rsaKey = generateRsa();
    JWKSet jwkSet = new JWKSet(rsaKey);
    return (jwkSelector, securityContext) -> jwkSelector.select(jwkSet);
}

private static RSAKey generateRsa() throws NoSuchAlgorithmException {
    KeyPair keyPair = generateRsaKey();
```

¹ <https://datatracker.ietf.org/doc/html/rfc7517>.

```

RSAPublicKey publicKey = (RSAPublicKey) keyPair.getPublic();
RSAPrivateKey privateKey = (RSAPrivateKey) keyPair.getPrivate();
return new RSAKey.Builder(publicKey)
    .privateKey(privateKey)
    .keyID(UUID.randomUUID().toString())
    .build();
}

private static KeyPair generateRsaKey() throws NoSuchAlgorithmException {
    KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance("RSA");
    keyPairGenerator.initialize(2048);
    return keyPairGenerator.generateKeyPair();
}

@Bean
public JwtDecoder jwtDecoder(JWKSource<SecurityContext> jwkSource) {
    return OAuth2AuthorizationServerConfiguration.jwtDecoder(jwkSource);
}

```

Давайте разберем, что делает этот код: `JWKSource` создает пары 2048-битных ключей RSA, которые будут использоваться для подписи токена. Токен подписывается с использованием закрытого ключа. Сервер ресурсов сможет проверить достоверность токена, указанного в запросе, получив открытый ключ от сервера авторизации. Эту процедуру мы рассмотрим подробнее, когда приступим к созданию сервера ресурсов.

Теперь наш сервер авторизации готов, осталось только запустить и опробовать его. После сборки и запуска приложения у вас появится сервер авторизации, прослушивающий порт 9000.

У нас пока нет клиента, поэтому используем веб-браузер и инструмент командной строки `curl`. Для начала введем в адресной строке веб-браузера URL `http://localhost:9000/oauth2/authorize?response_type=code&client_id=taco-admin-client&redirect_uri=http://127.0.0.1:9090/login/oauth2/code/taco-admin-client&-scope=writeIngredients+deleteIngredients1`. В результате должна открыться страница, как показано на рис. 8.2.

После входа (с учетными данными «tacochef» и «password» или любыми другими, хранящимися в базе данных `TacoUserRepository`) вам будет предложено дать согласие на доступ к запрошенным областям действия, как показано на рис. 8.3.

После получения согласия браузер будет переадресован на URL клиента. У нас пока нет клиента, поэтому, скорее всего, вы увидите сообщение об ошибке. Но это нормально – мы лишь имитируем работу клиента, поэтому сами получим код авторизации из URL.

¹ Обратите внимание, что этот и все другие URL в данной главе используют схему «`http://`». Это упрощает разработку и тестирование на локальной машине. Но в промышленном окружении всегда следует использовать более безопасную схему «`https://`».

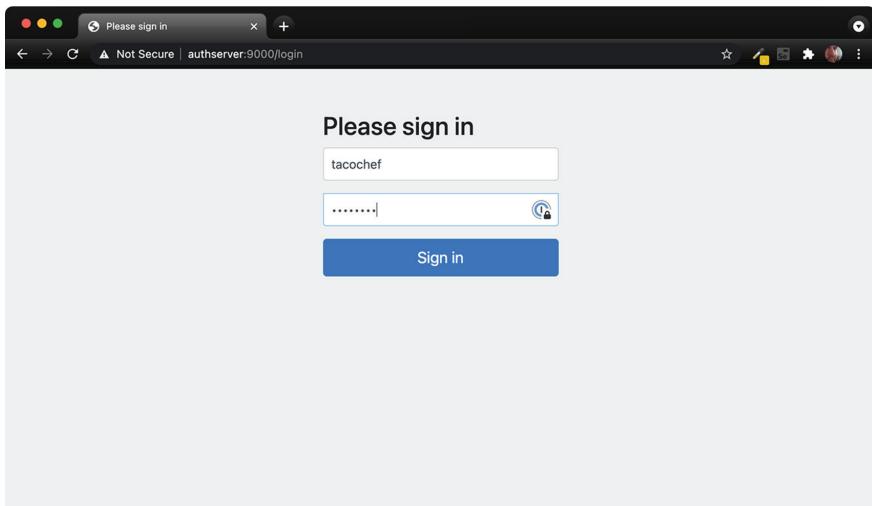


Рис. 8.2 Страница входа, сформированная сервером авторизации

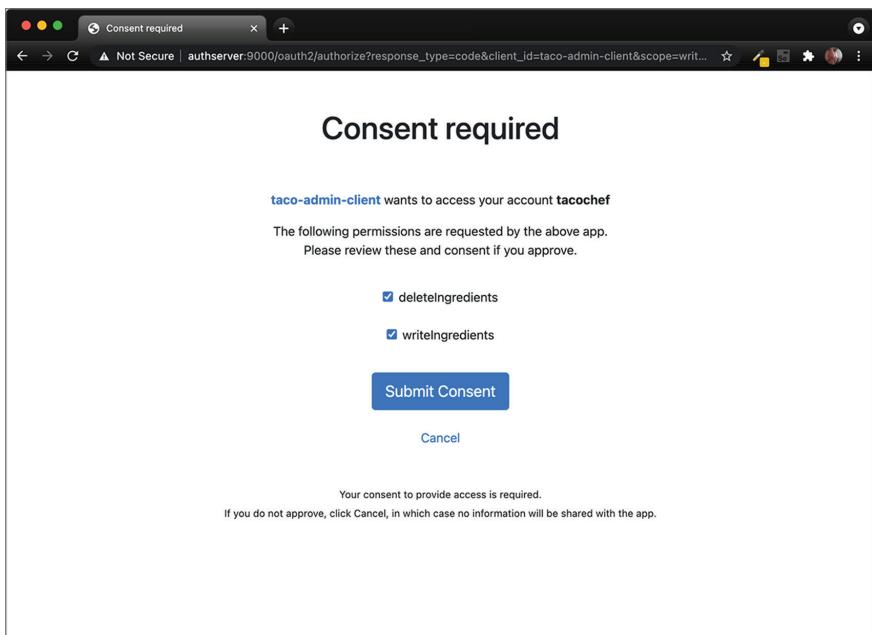


Рис. 8.3 Страница сервера авторизации с запросом согласия

Посмотрите на адресную строку браузера. Вы увидите, что URL имеет параметр `code`. Скопируйте значение этого параметра и вставьте его в следующую команду `curl` вместо `$code`:

```
$ curl localhost:9000/oauth2/token \
-H"Content-type: application/x-www-form-urlencoded" \
```

```
-d"grant_type=authorization_code" \
-d"redirect_uri=http://127.0.0.1:9090/login/oauth2/code/taco-admin-client"\ 
-d"code=$code" \
-u taco-admin-client:secret
```

Здесь мы обмениваем полученный код авторизации на токен доступа. Для тела запроса задается формат "application/x-www-form-urlencoded" и в самом запросе передаются: тип разрешения ("authorization_code"), URI переадресации (для дополнительной безопасности) и собственно код авторизации. Если все пройдет благополучно, то вы получите ответ JSON, который (в форматированном виде) выглядит так:

```
{
  "access_token": "eyJraWQ...",
  "refresh_token": "H0zHA5s...",
  "scope": "deleteIngredients writeIngredients",
  "token_type": "Bearer",
  "expires_in": "299"
}
```

Свойство "access_token" содержит токен доступа, который клиент может использовать для выполнения запросов к API. На самом деле он намного длиннее, чем показано здесь. Точно так же "refresh_token" был сокращен для экономии места. Теперь токен доступа можно отправлять в запросах серверу ресурсов, чтобы получить доступ к ресурсам, находящимся в областях "writeIngredients" и "deleteIngredients". Срок действия токена доступа ограничен 299 секундами (чуть меньше 5 минут), поэтому нужно действовать быстро, если мы собираемся его использовать. По истечении срока действия токена можно использовать токен обновления, чтобы получить новый токен доступа без повторного выполнения процесса авторизации.

Итак, как можно использовать токен доступа? Его можно отправить в запросе к Taco Cloud API в заголовке "Authorization", например так:

```
$ curl localhost:8080/ingredients \
-H"Content-type: application/json" \
-H"Authorization: Bearer eyJraWQ..." \
-d'{"id": "FISH", "name": "Stinky Fish", "type": "PROTEIN"}'
```

На данный момент токен ничего не дает нам. Это связано с тем, что Taco Cloud API пока не настроен на работу в режиме сервера ресурсов. Но мы все-таки можем проверить токен доступа, скопировав и вставив его в форму на <https://jwt.io>. Результат будет выглядеть примерно так, как показано на рис. 8.4.

Как видите, токен делится на три части: заголовок, данные и подпись. Взглянув на данные, можно заметить, что токен был выдан от имени пользователя «tacochef» и имеет области действия "writeIngredients" и "deleteIngredients". Как раз то, что мы просили!

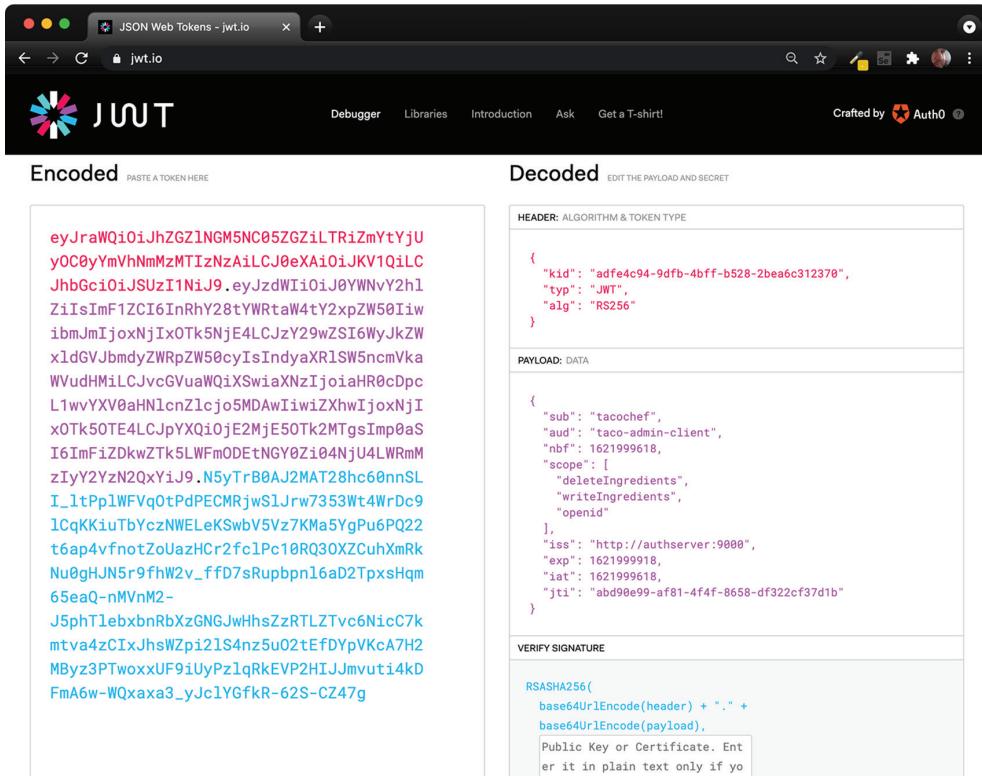


Рис. 8.4 Декодирование токена JWT на jwt.io

Примерно через 5 минут срок действия токена истечет. Но мы все еще можем проверить его в отладчике на <https://jwt.io>. Однако если передать его в реальном запросе к API, то он будет отклонен. В таком случае мы можем запросить новый токен доступа без повторного прохождения процедуры получения кода авторизации, нужно лишь послать новый запрос на сервер авторизации, используя разрешение "refresh_token" и передав токен обновления в параметре "refresh_token". Вот как можно сделать это с помощью curl:

```
$ curl localhost:9000/oauth2/token \
-H "Content-type: application/x-www-form-urlencoded" \
-d "grant_type=refresh_token&refresh_token=H0zHA5s..." \
-u taco-admin-client:secret
```

На этот запрос мы получим такой же ответ, как и на первый запрос, с помощью которого мы обменяли код авторизации на токен доступа, только с новым токеном доступа.

Вставлять токены доступа в форму <https://jwt.io> забавно, но истинное предназначение токена доступа – получить доступ к API. Поэтому давайте посмотрим, как добавить сервер ресурсов в Taco Cloud API.

8.3 Защита API с помощью сервера ресурсов

Сервер ресурсов на самом деле является просто фильтром перед API, проверяющим наличие действительного токена доступа с необходимой областью действия в запросах к ресурсам, требующим авторизации. Spring Security предоставляет свою реализацию сервера ресурсов OAuth2, которую можно добавить в существующий API, включив следующую зависимость в спецификацию сборки:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

Добавить сервер ресурсов можно также, выбрав зависимость **OAuth2 Resource Server** (Сервер ресурсов OAuth2) в Initializr при создании проекта.

После добавления зависимости следующим шагом нужно объявить, что для обработки запросов POST к конечной точке `/ingredients` требуется область `"writeIngredients"`, а для обработки запросов DELETE к `/ingredients` – область `"deleteIngredients"`. Следующий фрагмент кода из класса `SecurityConfig` проекта показывает, как это сделать:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .authorizeRequests()
        ...
        .antMatchers(HttpMethod.POST, "/api/ingredients")
        .hasAuthority("SCOPE_writeIngredients")
        .antMatchers(HttpMethod.DELETE, "/api//ingredients")
        .hasAuthority("SCOPE_deleteIngredients")
        ...
}
```

Для каждой из конечных точек вызывается метод `hasAuthority()`, чтобы задать требуемую область действия. Обратите внимание, что области имеют префикс `"SCOPE_"`, потому что они должны соответствовать областям OAuth 2, указанным в токенах доступа в запросах к этим ресурсам.

В том же классе конфигурации мы должны активизировать сервер ресурсов:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        ...
        .and()
        .oauth2ResourceServer(oauth2 -> oauth2.jwt())
        ...
}
```

Здесь в вызов `oauth2ResourceServer()` передается лямбда-выражение, настраивающее сервер ресурсов. В данном случае оно просто включает поддержку токенов JWT, чтобы сервер ресурсов мог проверить содержимое токена и определить имеющиеся разрешения. В частности, он будет проверять наличие в токене области "writeIngredients" и/или "deleteIngredients" в двух конечных точках, которые мы защищали.

Однако сервер ресурсов не будет слепо доверять самому факту наличия токена. Чтобы убедиться, что токен действительно создан доверенным сервером авторизации от имени пользователя, он проверит подпись токена с помощью открытого ключа, соответствующего закрытому ключу, который использовался для создания подписи токена. Однако мы должны подсказать серверу ресурсов, где получить открытый ключ. Следующее свойство определяет URL сервера авторизации, обратившись к которому, сервер ресурсов сможет получать открытый ключ:

```
spring:  
  security:  
    oauth2:  
      resourceserver:  
        jwt:  
          jwk-set-uri: http://localhost:9000/oauth2/jwks
```

Теперь сервер ресурсов готов к работе! Соберите и запустите приложение Taco Cloud. Затем попробуйте выполнить запрос с помощью `curl`:

```
$ curl localhost:8080/ingredients \  
  -H "Content-type: application/json" \  
  -d'{"id":"CRKT", "name":"Legless Crickets", "type":"PROTEIN"}'
```

Запрос должен завершиться ошибкой с кодом ответа HTTP 401. Это связано с тем, что теперь конечная точка требует наличия области действия "writeIngredients", а мы не передали в запросе действительного токена доступа с этой областью действия.

Чтобы запрос выполнился успешно и добавил новый ингредиент в базу данных, нужно получить токен доступа, используя процесс, описанный в предыдущем разделе, и указав требуемые области действия "writeIngredients" и "deleteIngredients". Затем поместить токен доступа в заголовок "Authorization", как показано в примере с утилитой `curl` ниже (замените "\$token" фактическим токеном доступа):

```
$ curl localhost:8080/ingredients \  
  -H "Content-type: application/json" \  
  -d'{"id":"SHMP", "name":"Coconut Shrimp", "type":"PROTEIN"}' \  
  -H "Authorization: Bearer $token"
```

На этот раз запрос должен завершиться успехом и добавить новый ингредиент. Проверить это можно, выполнив запрос GET к конечной точке `/ingredients`:

```
$ curl localhost:8080/ingredients
[
  {
    "id": "FLTO",
    "name": "Flour Tortilla",
    "type": "WRAP"
  },
  ...
  {
    "id": "SHMP",
    "name": "Coconut Shrimp",
    "type": "PROTEIN"
  }
]
```

Как видите, кокосовые креветки (Coconut Shrimp) теперь включены в конец списка всех ингредиентов, который вернула конечная точка */ingredients*. Отлично!

Напомню, что срок действия токена доступа ограничен 5 минутами. По истечении этого срока запросы снова начнут возвращать ответ HTTP 401. В таком случае можно получить новый токен доступа, отправив запрос на сервер авторизации, используя токен обновления, полученный вместе с токеном доступа (замените "\$refreshToken" фактическим токеном обновления):

```
$ curl localhost:9000/oauth2/token \
-H "Content-type: application/x-www-form-urlencoded" \
-d "grant_type=refresh_token&refresh_token=$refreshToken" \
-u taco-admin-client:secret
```

Снова созданным токеном доступа можно продолжать создавать новые ингредиенты сколько душе угодно.

Теперь, защитив конечную точку */ingredients*, стоит аналогичным образом защитить другие конечные точки, через которые может происходить утечка конфиденциальных данных. Например, конечная точка */orders*, вероятно, тоже должна быть закрыта для любых запросов, даже запросов GET, потому что позволяет получить информацию о клиенте. Я оставляю защиту этой и других конечных точек вам как самостоятельное упражнение.

Утилита `curl` вполне подходит для администрирования приложения Taco Cloud и для экспериментов с токенами OAuth 2. Но рано или поздно нам понадобится полноценное клиентское приложение, которое можно использовать для управления ингредиентами. Поэтому далее я предлагаю заняться созданием клиента с поддержкой OAuth, который будет получать токены доступа и отправлять запросы к API.

8.4 Разработка клиента

В процессе авторизации с использованием OAuth 2 клиентскому приложению отводится роль получения токена доступа и выполнения запросов к серверу ресурсов от имени пользователя. То есть когда клиентское приложение обнаруживает, что пользователь не прошел аутентификацию, оно должно переадресовать браузер на сервер авторизации, чтобы получить согласие от пользователя. Затем, когда сервер авторизации вернет управление клиенту, клиент должен обменять полученный код авторизации на токен доступа.

Такому приложению обязательно потребуется поддержка клиента Spring Security OAuth 2. Добавить подобную поддержку можно с помощью начальной зависимости:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

Она не только придаст приложению поддержку возможностей клиента OAuth 2, которыми мы вскоре воспользуемся, но также транзитивно добавит саму библиотеку Spring Security. Это позволит нам написать некоторую конфигурацию безопасности для приложения. Следующий bean-компонент `SecurityFilterChain` настраивает Spring Security так, чтобы все запросы требовали аутентификации:

```
@Bean
SecurityFilterChain defaultSecurityFilterChain(HttpSecurity http) throws
    Exception {
    http
        .authorizeRequests(
            authorizeRequests -> authorizeRequests.anyRequest().authenticated()
        )
        .oauth2Login(
            oauth2Login ->
                oauth2Login.loginPage("/oauth2/authorization/taco-admin-client")
                .oauth2Client(withDefaults());
        )
    return http.build();
}
```

Более того, компонент `SecurityFilterChain` также включает поддержку OAuth 2 на стороне клиента. В частности, он настраивает путь к странице входа `/oauth2/authorization/taco-admin-client`. Но это не обычная страница входа, которая запрашивает имя пользователя и пароль. Она принимает код авторизации, обменивает его на токен доступа и использует токен доступа для идентификации пользователя. Иначе говоря, это URL, куда сервер авторизации передаст управление, когда пользователь даст разрешение.

Нам также необходимо задать настройки сервера авторизации и информацию о клиенте OAuth 2 нашего приложения. Это делается в конфигурационных свойствах, например в файле *application.yml*, который настраивает клиента с именем `taco-admin-client`:

```
spring:
  security:
    oauth2:
      client:
        registration:
          taco-admin-client:
            provider: tacocloud
            client-id: taco-admin-client
            client-secret: secret
            authorization-grant-type: authorization_code
            redirect-uri:
              - "http://127.0.0.1:9090/login/oauth2/code/{registrationId}"
            scope: writeIngredients,deleteIngredients,openid
```

Эта конфигурация регистрирует клиента Spring Security OAuth 2 с именем `taco-admin-client`. Регистрационная информация включают учетные данные клиента (`client-id` и `client-secret`), тип разрешения (`authorization-grant-type`), запрашиваемые области действий (`scope`) и URI для переадресации (`redirect-uri`). Обратите внимание, что значение, указанное в `redirect-uri`, включает заполнитель, ссылающийся на идентификатор регистрации клиента (`registrationId`), то есть `taco-admin-client`. Соответственно, фактический URI переадресации будет иметь тот же путь, который мы настроили выше и ссылающийся на страницу входа OAuth 2: `http://127.0.0.1:9090/login/oauth2/code/taco-admin-client`.

А как же сам сервер авторизации? Как сообщить клиенту, что он должен переадресовать браузер пользователю? За это отвечает свойство `provider`, хотя и косвенно. Свойству `provider` присвоено значение `tacocloud`, которое является ссылкой на отдельный набор настроек, описывающей сервер авторизации провайдера `tacocloud`. Конфигурация этого провайдера настраивается в том же файле *application.yml*:

```
spring:
  security:
    oauth2:
      client:
        ...
      provider:
        tacocloud:
          issuer-uri: http://authserver:9000
```

Единственное свойство, необходимое для настройки провайдера, – `issuer-uri`. Это свойство определяет базовый URI сервера авторизации. В данном случае он задает имя хоста сервера `authserver`. Учитывая, что мы запускаем эти примеры локально, это просто еще один

псевдоним для `localhost`. В большинстве операционных систем Unix это имя можно добавить в файл `/etc/hosts`:

```
127.0.0.1 authserver
```

За более подробной информацией о настройке разрешения сетевых имен, если у вас прием с `/etc/hosts` не работает, обращайтесь к документации по вашей операционной системе.

Основываясь на базовом URL, клиент Spring Security OAuth 2 выберет разумные значения по умолчанию для URL авторизации, URL токена и других характеристик сервера авторизации. Но если по какой-то причине настройки сервера авторизации, с которым вы работаете, отличаются от этих значений по умолчанию, то можно явно настроить детали авторизации:

```
spring:
  security:
    oauth2:
      client:
        provider:
          tacocloud:
            issuer-uri: http://authserver:9000
            authorization-uri: http://authserver:9000/oauth2/authorize
            token-uri: http://authserver:9000/oauth2/token
            jwk-set-uri: http://authserver:9000/oauth2/jwks
            user-info-uri: http://authserver:9000/userinfo
            user-name-attribute: sub
```

Мы уже видели большинство этих URI, в том числе URI авторизации, URI токена и URI JWK Set. Однако свойство `user-info-uri` мы еще не встречали. Этот URI используется клиентом для получения важной информации о пользователе, в первую очередь имени пользователя. Запрос к этому URI должен возвращать ответ JSON, включающий свойство, указанное в `user-name-attribute`, для идентификации пользователя. Однако обратите внимание, что при использовании Spring Authorization Server нет необходимости создавать конечную точку для этого URI; Spring Authorization Server автоматически создаст ее.

Теперь в приложении есть все необходимое для аутентификации и получения токена доступа от сервера авторизации. Сейчас можно запустить приложение, послать запрос на любой URL, и приложение автоматически переадресует вас на сервер авторизации. Когда сервер авторизации переадресует браузер обратно, клиентская библиотека Spring Security OAuth 2 обменяет полученный код авторизации на токен доступа. Но как использовать этот токен?

Предположим, у нас есть компонент, взаимодействующий с Taco Cloud API посредством `RestTemplate`. Следующая реализация `RestIngredientService` показывает класс, предлагающий два метода: один для получения списка ингредиентов и другой для сохранения нового ингредиента:

```

package tacos;

import java.util.Arrays;
import org.springframework.web.client.RestTemplate;

public class RestIngredientService implements IngredientService {

    private RestTemplate restTemplate;

    public RestIngredientService() {
        this.restTemplate = new RestTemplate();
    }

    @Override
    public Iterable<Ingredient> findAll() {
        return Arrays.asList(restTemplate.getForObject(
            "http://localhost:8080/api/ingredients",
            Ingredient[].class));
    }

    @Override
    public Ingredient addIngredient(Ingredient ingredient) {
        return restTemplate.postForObject(
            "http://localhost:8080/api/ingredients",
            ingredient,
            Ingredient.class);
    }
}

```

Защита конечной точки */ingredients* не затрагивает HTTP-запросы GET к ней, поэтому метод `findAll()` должен работать как обычно, если Taco Cloud API прослушивает порт 8080 локального хоста. Но метод `addIngredient()`, скорее всего, не будет работать, получая ответ HTTP 401, потому что мы настроили защиту */ingredients* от POST-запросов и потребовали, чтобы вместе с запросом посыпался токен доступа с областью действия `"writeIngredients"`. Единственный способ обеспечить обработку запросов – отправлять вместе с ними соответствующий токен доступа в заголовке `Authorization`.

К счастью, клиент Spring Security OAuth 2 уже должен иметь токен доступа после завершения процесса авторизации. От нас требуется только добавить этот токен в запрос. Для этого изменим конструктор и подключим в нем перехватчик запросов к `RestTemplate`:

```

public RestIngredientService(String accessToken) {
    this.restTemplate = new RestTemplate();
    if (accessToken != null) {
        this.restTemplate
            .getInterceptors()
            .add(getBearerTokenInterceptor(accessToken));
    }
}

```

```

private ClientHttpRequestInterceptor
    getBearerTokenInterceptor(String accessToken) {
    ClientHttpRequestInterceptor interceptor =
        new ClientHttpRequestInterceptor() {
    @Override
    public ClientHttpResponse intercept(
        HttpRequest request, byte[] bytes,
        ClientHttpRequestExecution execution) throws IOException {
        request.getHeaders().add("Authorization", "Bearer " + accessToken);
        return execution.execute(request, bytes);
    }
};
return interceptor;
}

```

Конструктор теперь принимает параметр типа `String` – токен доступа. Используя этот токен, он присоединяет перехватчик клиентских запросов, который добавляет заголовок `Authorization` к каждому запросу, посылаемому компонентом `RestTemplate`, со значением `"Bearer"`, за которым следует значение токена. Для поддержания порядка в конструкторе клиентский перехватчик создается в отдельном приватном вспомогательном методе.

Остается только один вопрос: откуда берется токен доступа? Все волшебство творится в следующем методе:

```

@Bean
@RequestScope
public IngredientService ingredientService(
    OAuth2AuthorizedClientService clientService) {
    Authentication authentication =
        SecurityContextHolder.getContext().getAuthentication();

    String accessToken = null;

    if (authentication.getClass()
        .isAssignableFrom(OAuth2AuthenticationToken.class)) {
        OAuth2AuthenticationToken oauthToken =
            (OAuth2AuthenticationToken) authentication;
        String clientRegistrationId =
            oauthToken.getAuthorizedClientRegistrationId();
        if (clientRegistrationId.equals("taco-admin-client")) {
            OAuth2AuthorizedClient client =
                clientService.loadAuthorizedClient(
                    clientRegistrationId, oauthToken.getName());
            accessToken = client.getAccessToken().getTokenValue();
        }
    }
    return new RestIngredientService(accessToken);
}

```

Для начала обратите внимание, что с помощью аннотации `@RequestScope` bean-компонент объявлен как принадлежащий области

видимости запроса. Это означает, что для каждого запроса будет создаваться новый экземпляр компонента. Компонент должен принадлежать области видимости запроса, потому что ему необходимо получить информацию об аутентификации из `SecurityContext`, который заполняется при каждом запросе одним из фильтров Spring Security; эта информация отсутствует в `SecurityContext` во время запуска приложения, когда создаются bean-компоненты с областью видимости по умолчанию.

Перед возвратом экземпляра `RestIngredientService` метод проверяет, действительно ли аутентификация реализована как `OAuth2AuthenticationToken`. Если да, то это означает, что у него будет токен. Затем метод проверяет, предназначен ли токен аутентификации для клиента с именем `taco-admin-client`. Если это так, то он извлекает токен из авторизованного клиента и передает его в конструктор `RestIngredientService`. Получив токен, `RestIngredientService` не будет испытывать проблем с выполнением запросов к конечным точкам Taco Cloud API от имени пользователя, авторизовавшего приложение.

Итоги

- Безопасность OAuth 2 – это распространенный способ защиты API, более надежный, чем простая аутентификация HTTP Basic.
- Сервер авторизации выпускает токены доступа, позволяющие клиенту выполнять запросы к API от имени пользователя (или от своего имени, если используются клиентские токены).
- Сервер ресурсов находится перед API и осуществляет проверку действительности токенов перед передачей запросов ресурсам.
- Spring Authorization Server – это экспериментальный проект, реализующий сервер авторизации OAuth 2.
- Spring Security поддерживает создание сервера ресурсов, а также создание клиентов, которые получают токены доступа от сервера авторизации и передают их в запросах серверу ресурсов.

Асинхронная передача сообщений

В этой главе рассматриваются следующие темы:

- асинхронная передача сообщений;
- передача сообщений с помощью JMS, RabbitMQ и Kafka;
- получение сообщений из брокера;
- пассивный прием сообщений.

Сейчас на часах 16:55, пятница. Вы в нескольких минутах от начала долгожданного отпуска. У вас достаточно времени, чтобы доехать до аэропорта и успеть на свой рейс. Но, прежде чем собраться и отправиться в путь, вы должны убедиться, что ваш руководитель и коллеги знают, на каком этапе находится ваш проект, чтобы в понедельник они могли продолжить с того места, на котором вы остановились. К сожалению, некоторые из ваших коллег уже ушли на выходные, а ваш начальник занят на совещании. Что же делать?

Лучший способ сообщить необходимую информацию и успеть на самолет – отправить электронное письмо начальнику и коллегам, подробно описав свои успехи и пообещав прислать открытку. Вы не знаете, где они находятся и когда прочитают ваше письмо, но вы знаете, что рано или поздно они сядут за свои столы и прочитают письмо. А вы тем временем направляетесь в аэропорт.

Синхронные взаимодействия, которые мы наблюдали в REST, имеют свою нишу. Но это не единственный способ обмена информацией между приложениями. Асинхронный обмен сообщениями – это способ

косвенной отправки сообщений из одного приложения в другое, не предполагающий ожидания ответа. Такая косвенность обеспечивает более слабую связь и лучшую масштабируемость между взаимодействующими приложениями.

В этой главе мы используем приемы асинхронного обмена сообщениями для отправки заказов с веб-сайта Taco Cloud в отдельное приложение на кухнях Taco Cloud, где будут готовиться тако. Мы рассмотрим три варианта асинхронного обмена сообщениями, поддерживаемых в Spring: Java Message Service (JMS), RabbitMQ с протоколом Advanced Message Queuing Protocol (AMQP) и Apache Kafka. Помимо отправки и получения простых сообщений, мы познакомимся также с поддержкой в Spring объектов Java, управляемых сообщениями: способ получения сообщений, напоминающий использование компонентов, управляемых сообщениями (Message-Driven Beans, MDB) в Enterprise JavaBeans.

9.1 Отправка сообщений с помощью JMS

Служба обмена сообщениями (Java Message Service, JMS) – это стандарт Java, определяющий общий API для работы с брокерами сообщений. Впервые представленная в 2001 году, JMS долгое время оставалась основным решением асинхронного обмена сообщениями в Java. До JMS у каждого брокера сообщений был свой API, что усложняло реализацию совместимости с разными брокерами. JMS обеспечила возможность работы со всеми совместимыми реализациями через общий интерфейс почти так же, как JDBC обеспечила общий интерфейс к реляционным базам данных.

В Spring поддержка JMS основана на абстракции, известной как `JmsTemplate`, которая позволяет отправителю посыпать сообщения в разные очереди и темы, а получателю принимать их. Spring также поддерживает понятие простых объектов Java (Plain Old Java Object, POJO), управляемых сообщениями, которые асинхронно обрабатывают сообщения, поступающие в очередь или тему.

Далее мы исследуем поддержку JMS в Spring, включая `JmsTemplate` и объекты, управляемые сообщениями. Но все наше внимание будет сосредоточено именно на поддержке, имеющейся в Spring. Если вы захотите узнать больше о JMS, то я рекомендую прочитать книгу Брюса Снайдера (Bruce Snyder), Деяна Босанака (Dejan Bosanac) и Роба Дэвиса (Rob Davies) «*ActiveMQ in Action*» (Manning, 2011).

Прежде чем начать отправлять и получать сообщения, нужно создать брокера сообщений, готового переносить эти сообщения между отправителями и получателями. Поэтому начнем наше знакомство с Spring JMS с настройки брокера сообщений.

9.1.1 Настройка JMS

Чтобы получить возможность использовать JMS, нужно добавить клиента JMS в спецификацию сборки проекта. При использовании Spring Boot это проще простого, нужно лишь добавить начальную зависимость в спецификацию сборки. Однако сначала нужно решить, какой брокер будет использоваться – Apache ActiveMQ или более новый Apache ActiveMQ Artemis.

Если вы выберете ActiveMQ, то добавьте следующую зависимость в файл *pom.xml* проекта:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

Если ваш выбор падет на ActiveMQ Artemis, то добавьте эту зависимость:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-artemis</artifactId>
</dependency>
```

При создании проекта с помощью Spring Initializr (или интерфейса к Initializr в вашей IDE) также можно выбрать любой из доступных параметров – «Spring for Apache ActiveMQ 5» и «Spring for Apache ActiveMQ Artemis», – как показано на рис. 9.1, где изображен фрагмент страницы со списком зависимостей на сайте <https://start.spring.io>.

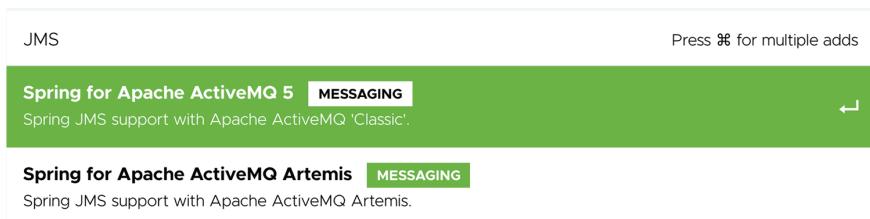


Рис. 9.1 Варианты ActiveMQ и Artemis, доступные в Spring Initializr

Artemis – это обновленная реализация ActiveMQ, фактически пре-вращающая ActiveMQ в устаревший вариант. Поэтому для Taco Cloud выберем Artemis. Но, вообще говоря, этот выбор мало повлияет на наш код, отправляющий и получающий сообщения. Единственное существенное различие заключается в настройках соединений с бро-кером.

ЗАПУСК БРОКЕРА ARTEMIS Для опробования примеров кода, представленных в этой главе, вам понадобится действующий брокер Artemis. Если у вас пока нет запущенного экземпляра Artemis, то обращайтесь за инструкциями в документации Artemis, которую можно найти по адресу <http://mng.bz/Xr81>.

По умолчанию Spring предполагает, что брокер Artemis прослушивает порт 61616 на локальном хосте. Эти настройки прекрасно подходят для разработки, но перед развертыванием приложения в промышленном окружении вам нужно будет настроить несколько свойств, описывающих детали доступа к брокеру. Они перечислены в табл. 9.1.

Таблица 9.1 Свойства, определяющие параметры доступа к брокеру Artemis

Свойство	Описание
spring.artemis.host	Хост брокера
spring.artemis.port	Порт брокера
spring.artemis.user	Имя пользователя для доступа к брокеру (необязательно)
spring.artemis.password	Пароль для доступа к брокеру (необязательно)

Например, взгляните на следующую запись в файле *application.yml*, определяющую настройки для промышленного окружения:

```
spring:
  artemis:
    host: artemis.tacocloud.com
    port: 61617
    user: tacoweb
    password: l3tm31n
```

Этот фрагмент определяет настройки Spring для подключения к брокеру Artemis, действующему на хосте `artemis.tacocloud.com` и прослушивающему порт 61617. Здесь также определены учетные данные для приложения, которое будет взаимодействовать с этим брокером. Учетные данные необязательны, но рекомендуются для случаев промышленного использования.

Если бы мы решили использовать ActiveMQ вместо Artemis, то нам пришлось бы настроить свойства, специфические для ActiveMQ, перечисленные в табл. 9.2.

Таблица 9.2 Свойства, определяющие параметры доступа к брокеру ActiveMQ

Свойство	Описание
spring.activemq.broker-url	URL брокера
spring.activemq.user	Имя пользователя для доступа к брокеру (необязательно)
spring.activemq.password	Пароль для доступа к брокеру (необязательно)
spring.activemq.in-memory	Признак размещения запуска брокера в памяти процесса (по умолчанию: <code>true</code>)

Обратите внимание, что вместо отдельных свойств, определяющих имя хоста и порт брокера, ActiveMQ поддерживает одно общее свойство `spring.activemq.broker-url`, в котором нужно указать URL со схемой `tcp://`, как показано в следующем фрагменте кода YAML:

```
spring:  
  activemq:  
    broker-url: tcp://activemq.tacocloud.com  
    user: tacoweb  
    password: l3tm31n
```

Независимо от выбранного варианта – Artemis или ActiveMQ – в окружении разработки, когда брокер работает локально, эти свойства настраивать не нужно.

Но если вы используете ActiveMQ, то установите свойство `spring.activemq.in-memory` в значение `false`, чтобы Spring не запускал брокера в памяти процесса. Запуск брокера в памяти процесса может пригодиться, только когда обмен сообщениями происходит исключительно в рамках одного приложения.

Прежде чем двинуться дальше, мы должны запустить внешний экземпляр брокера Artemis (или ActiveMQ). Но я не буду повторять здесь инструкции, которые вы найдете в документации к брокерам:

- *Artemis* – <http://mng.bz/yJOo>;
- *ActiveMQ* – <http://mng.bz/MveD>.

После добавления начальной зависимости JMS и запуска брокера, готового передавать сообщения из одного приложения в другое, можно начинать отправлять сообщения.

9.1.2 Отправка сообщений с помощью `JmsTemplate`

Благодаря включению начальной зависимости JMS (Artemis или ActiveMQ) в спецификацию сборки Spring Boot автоматически настроит механизм `JmsTemplate`, который вы сможете внедрять и использовать для отправки и получения сообщений.

`JmsTemplate` – центральный элемент интеграции Spring с JMS. Подобно другим компонентам Spring, ориентированным на шаблоны, `JmsTemplate` избавляет от необходимости писать большой объем шаблонного кода. Без `JmsTemplate` вам придется написать код, открывающий соединение с брокером сообщений и создающий сеанс, а также дополнительный код для обработки любых исключений, которые могут возникнуть в процессе отправки сообщения. `JmsTemplate` позволяет сосредоточиться на том, что вы действительно хотите сделать: на отправке сообщений.

`JmsTemplate` имеет несколько методов отправки сообщений, в том числе:

```
// Отправка неструктурированных сообщений  
void send(MessageCreator messageCreator) throws JmsException;  
void send(Destination destination, MessageCreator messageCreator)
```

```

throws JmsException;
void send(String destinationName, MessageCreator messageCreator)
throws JmsException;

// Отправка сообщений в форме объектов
void convertAndSend(Object message) throws JmsException;
void convertAndSend(Destination destination, Object message)
throws JmsException;
void convertAndSend(String destinationName, Object message)
throws JmsException;

// Отправка сообщений в форме объектов с постобработкой
void convertAndSend(Object message,
MessagePostProcessor postProcessor) throws JmsException;
void convertAndSend(Destination destination, Object message,
MessagePostProcessor postProcessor) throws JmsException;
void convertAndSend(String destinationName, Object message,
MessagePostProcessor postProcessor) throws JmsException;

```

Как видите, на самом деле есть только два метода, `send()` и `convertAndSend()`, каждый из которых имеет перегруженные версии, поддерживающие разные наборы параметров. И если присмотреться, то можно заметить, что разные формы `convertAndSend()` можно разделить на две подкатегории:

- три метода `send()` принимают `MessageCreator` для создания объекта `Message`;
- три метода `convertAndSend()` принимают объект `Object` и автоматически преобразуют его в сообщение;
- три метода `convertAndSend()` автоматически преобразуют объект `Object` в сообщение, а также принимают `MessagePostProcessor` для дополнительной настройки сообщения `Message` перед его фактической отправкой.

Кроме того, каждая из этих категорий включает три перегруженные версии метода, которые различаются способом определения места назначения в JMS (очередь или тема), а именно:

- один метод не принимает никаких параметров и отправляет сообщение в место назначения по умолчанию;
- один метод принимает объект `Destination`, определяющий место назначения для сообщения;
- один метод принимает строку, указывающую имя места назначения для сообщения.

В листинге 9.1 приводится пример реализации службы `JmsOrderMessagingService`, использующей самую простую версию метода `send()`.

Листинг 9.1 Отправка заказа с помощью метода `send()` в место назначения по умолчанию

```

package tacos.messaging;
import javax.jms.JMSException;
import javax.jms.Message;

```

```

import javax.jms.Session;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;
import org.springframework.stereotype.Service;

@Service
public class JmsOrderMessagingService implements OrderMessagingService {

    private JmsTemplate jms;

    @Autowired
    public JmsOrderMessagingService(JmsTemplate jms) {
        this.jms = jms;
    }

    @Override
    public void sendOrder(TacoOrder order) {
        jms.send(new MessageCreator() {
            @Override
            public Message createMessage(Session session)
                    throws JMSException {
                return session.createObjectMessage(order);
            }
        });
    }
}

```

Метод `sendOrder()` вызывает `jms.send()` и передает анонимную реализацию `MessageCreator`, объявленную внутри класса. Эта реализация переопределяет метод `createMessage()`, создающий новый объект сообщения из данного объекта `TacoOrder`.

Поскольку служба `JmsOrderMessagingService` ориентирована на использование JMS и реализует более общий интерфейс `OrderMessagingService`, мы можем внедрить ее в `OrderApiController` и вызывать ее метод `sendOrder()` при создании заказа:

```

@RestController
@RequestMapping(path="/api/orders",
        produces="application/json")
@CrossOrigin(origins="http://localhost:8080")
public class OrderApiController {

    private OrderRepository repo;
    private OrderMessagingService messageService;

    public OrderApiController(
            OrderRepository repo,
            OrderMessagingService messageService) {
        this.repo = repo;
        this.messageService = messageService;
    }
}

```

```

    }

    @PostMapping(consumes="application/json")
    @ResponseStatus(HttpStatus.CREATED)
    public TacoOrder postOrder(@RequestBody TacoOrder order) {
        messageService.sendOrder(order);
        return repo.save(order);
    }

    ...
}

}

```

Теперь, сразу после создания заказа на веб-сайте Taco Cloud, брокеру должно быть отправлено сообщение для доставки другому приложению, которое получит заказ. У нас пока нет такого приложения. Однако мы можем для эксперимента использовать консоль Artemis и с ее помощью просмотреть содержимое очереди. Подробнее о том, как это сделать, рассказывается в документации Artemis, по адресу <http://mng.bz/aZx9>.

Не знаю, как вам, но мне кажется, что код в листинге 9.1 хоть и прост, но выглядит несколько неуклюжим. Объявление анонимного внутреннего класса усложняет простой вызов метода. Зная, что `MessageCreator` – это функциональный интерфейс, мы можем немного упростить метод `sendOrder()`, используя лямбда-выражение:

```

@Override
public void sendOrder(TacoOrder order) {
    jms.send(session -> session.createObjectMessage(order));
}

```

Но обратите внимание, что вызов `jms.send()` не определяет место назначения. Чтобы такой подход сработал, нужно также задать имя места назначения по умолчанию с помощью свойства `spring.jms.template.default-destination`. Например, вот как можно установить свойство в файле `application.yml`:

```

spring:
  jms:
    template:
      default-destination: tacocloud.order.queue

```

Во многих случаях использование места назначения по умолчанию является самым простым выбором. Этот прием позволяет указать имя места назначения один раз и в коде просто отправлять сообщения, куда бы они ни направлялись. Но если когда-нибудь понадобится отправить сообщение в место назначения, отличное от места назначения по умолчанию, то это место нужно будет указать в параметре `send()`.

Один из возможных вариантов – передать объект `Destination` в первом параметре. Самый простой способ сделать это – объявить

компонент `Destination`, а затем внедрить его в компонент, посылающий сообщения. Например, следующий компонент объявляет место назначения `Destination` для очереди заказов Taco Cloud:

```
@Bean
public Destination orderQueue() {
    return new ActiveMQQueue("tacocloud.order.queue");
}
```

Этот метод можно добавить в любой конфигурационный класс в приложении, которое будет отправлять или получать сообщения через JMS. Для более упорядоченной организации кода его лучше добавить в конфигурационный класс, предназначенный для настройки обмена сообщениями, например `MessagingConfig`.

Важно отметить, что `ActiveMQQueue`, используемый здесь, на самом деле принадлежит Artemis (находится в пакете `org.apache.activemq.artemis.jms.client`). Если вместо Artemis вы решите использовать ActiveMQ, то нужный вам класс с именем `ActiveMQQueue` будет находиться в пакете `org.apache.activemq.command`.

Внедрив компонент `Destination` в `JmsOrderMessagingService`, вы сможете использовать его при вызове `send()` следующим образом:

```
private Destination orderQueue;

@Autowired
public JmsOrderMessagingService(JmsTemplate jms,
                                  Destination orderQueue) {
    this.jms = jms;
    this.orderQueue = orderQueue;
}

...

@Override
public void sendOrder(TacoOrder order) {
    jms.send(
        orderQueue,
        session -> session.createObjectMessage(order));
}
```

Указание места назначения с помощью объекта `Destination` открывает более широкие возможности настройки. Но на практике редко бывает нужно настраивать что-то еще в месте назначения, кроме имени. Часто бывает проще указать имя в первом параметре `send()`, как показано здесь:

```
@Override
public void sendOrder(TacoOrder order) {
    jms.send(
        "tacocloud.order.queue",
        session -> session.createObjectMessage(order));
}
```

Метод `send()` относительно прост в использовании (особенно если `MessageCreator` задается как лямбда-выражение), но когда требуется предоставить полноценную реализацию `MessageCreator`, сложность заметно увеличивается. Было бы проще, если бы требовалось указать только объект, который нужно отправить (и, возможно, место назначения). Такую возможность дает метод `convertAndSend()`. Давайте посмотрим, как он используется.

ПРЕОБРАЗОВАНИЕ СООБЩЕНИЙ ПЕРЕД ОТПРАВКОЙ

Метод `convertAndSend()` упрощает публикацию сообщений, избавляя от необходимости предоставлять свою реализацию `MessageCreator`. Он позволяет просто передать объект, который нужно отправить, и этот объект автоматически будет преобразован в сообщение перед отправкой.

Например, следующая реализация `sendOrder()` использует `convertAndSend()` для отправки `TacoOrder` в указанное место назначения:

```
@Override
public void sendOrder(TacoOrder order) {
    jms.convertAndSend("tacocloud.order.queue", order);
}
```

Так же как `send()`, метод `convertAndSend()` принимает место назначения в виде объекта `Destination` или строки с именем места назначения. Также место назначения можно вообще не указывать, и тогда сообщение будет отправлено в место назначения по умолчанию.

Какую бы версию `convertAndSend()` вы ни выбрали, она автоматически преобразует `TacoOrder` в объект `Message` перед отправкой. Этот автоматизм обеспечивается реализацией `MessageConverter`, которая выполняет всю грязную работу по преобразованию объектов предметной области в объекты `Message`.

НАСТРОЙКА КОНВЕРТЕРА СООБЩЕНИЙ

`MessageConverter` – это интерфейс, определенный в Spring, который имеет только два метода:

```
public interface MessageConverter {
    Message toMessage(Object object, Session session)
        throws JMSException, MessageConversionException;
    Object fromMessage(Message message)
}
```

Этот интерфейс довольно прост в реализации, но писать свои реализации приходится довольно редко. Spring предлагает несколько готовых реализаций, перечисленных в табл. 9.3.

По умолчанию используется `SimpleMessageConverter`, но этот конвертер требует, чтобы отправляемый объект реализовал интерфейс `Serializable`. Это может быть неплохой выбор, но иногда предпочти-

тельнее использовать другие конвертеры, например `MappingJackson2MessageConverter`, чтобы избежать этого ограничения.

Таблица 9.3 Конвертеры сообщений в Spring для наиболее распространенных случаев применения (все они находятся в пакете `org.springframework.jms.support.converter`)

Конвертер сообщений	Описание
<code>MappingJackson2MessageConverter</code>	Использует библиотеку Jackson 2 JSON для преобразования сообщений в формат JSON и обратно
<code>MarshallingMessageConverter</code>	Использует JAXB для преобразования сообщений в формат XML и обратно
<code>MessagingMessageConverter</code>	Использует встроенную реализацию <code>MessageConverter</code> для преобразования тела сообщения из абстракции <code>Message</code> и обратно и <code>JmsHeaderMapper</code> – для отображения заголовков JMS в стандартные заголовки и обратно
<code>SimpleMessageConverter</code>	Преобразует <code>String</code> в <code>TextMessage</code> и обратно, массивы байтов в <code>BytesMessage</code> и обратно, <code>Map</code> в <code>MapMessage</code> и обратно и <code>Serializable</code> в <code>ObjectMessage</code> и обратно

Чтобы применить другой конвертер сообщений, нужно лишь объявить его экземпляр компонентом. Например, следующее объявление позволит использовать `MappingJackson2MessageConverter` вместо `SimpleMessageConverter`:

```
@Bean
public MappingJackson2MessageConverter messageConverter() {
    MappingJackson2MessageConverter messageConverter =
        new MappingJackson2MessageConverter();
    messageConverter.setTypeIdPropertyName("_ typeId");
    return messageConverter;
}
```

Этот метод можно поместить в любой конфигурационный класс в приложении, которое будет отправлять или получать сообщения через JMS, например в `MessagingConfig` вместе с компонентом `Destination`.

Обратите внимание, что здесь мы вызываем метод `setTypeIdPropertyName()` компонента `MappingJackson2MessageConverter` перед его возвратом. Это важный шаг, позволяющий получателю узнать, в какой тип нужно преобразовать входящее сообщение. По умолчанию это свойство содержит полное имя класса преобразуемого типа. Но данное решение недостаточно гибкое и требует, чтобы получатель также имел тот же тип с тем же полным именем класса.

Чтобы повысить гибкость, можно отобразить имя синтетического типа в фактический тип, вызвав метод `setTypeIdMappings()` конвертера сообщений. Например, следующее изменение отобразит идентификатор синтетического типа `TacoOrder` в класс `TacoOrder`:

```
@Bean
public MappingJackson2MessageConverter messageConverter() {
```

```

MappingJackson2MessageConverter messageConverter =
        new MappingJackson2MessageConverter();
messageConverter.setTypeIdPropertyName("_ typeId");

Map<String, Class<?>> typeIdMappings = new HashMap<String, Class<?>>();
typeIdMappings.put("order", TacoOrder.class);
messageConverter.setTypeIdMappings(typeIdMappings);

return messageConverter;
}

```

Вместо полного имени класса, отправляемого в свойстве `_ typeId` сообщения, будет отправлено значение `TacoOrder`. Приложение-получатель настроит аналогичный конвертер сообщений, отображающий `TacoOrder` в свое представление заказа. Это представление заказа может находиться в другом пакете, иметь другое имя и даже иметь лишь часть свойств типа `TacoOrder` на стороне отправителя.

ПОСТОБРАБОТКА СООБЩЕНИЙ

Предположим, что в дополнение к своему прибыльному веб-бизнесу в `Taco Cloud` решили открыть несколько традиционных закусочных. Учитывая, что любое из их заведений также может выполнять заказы, сделанные через интернет, нужен какой-то способ сообщать источник заказов, чтобы персонал заведений мог организовать разные процессы обработки заказов, сделанных посетителями заведения и через интернет.

Было бы разумно добавить новое свойство `source` в объект `TacoOrder` для передачи этой информации: `WEB` – для заказов, сделанных через интернет, и `STORE` – для заказов, сделанных посетителями заведения. Но для этого нужно внести изменения в класс `TacoOrder` веб-приложения и в класс `TacoOrder` кухонного приложения, хотя на самом деле эта информация нужна только тем, кто готовит тако.

Более простое решение – добавить в сообщение отдельный заголовок, указывающий источник заказа. При использовании метода `send()` для отправки сообщений с заказами это можно сделать вызовом метода `setStringProperty()` объекта `Message`:

```

jms.send("tacocloud.order.queue",
    session -> {
        Message message = session.createObjectMessage(order);
        message.setStringProperty("X_ORDER_SOURCE", "WEB");
    });

```

Проблема в том, что мы не используем `send()`. При использовании `convertAndSend()` объект `Message` создается скрытно, без нашего участия.

К счастью, у нас есть способ скорректировать экземпляр `Message`, созданный в недрах `convertAndSend()`, перед его отправкой. Пере-

дав `MessagePostProcessor` в последнем параметре `convertAndSend()`, можно организовать дополнительную обработку экземпляра сообщения после его создания. Следующий код все также использует `convertAndSend()`, но передает этому методу `MessagePostProcessor`, добавляющий заголовок `X_ORDER_SOURCE` в сообщение `Message` перед его отправкой:

```
jms.convertAndSend("tacocloud.order.queue", order, new MessagePostProcessor()
{
    @Override
    public Message postProcessMessage(Message message) throws JMSException {
        message.setStringProperty("X_ORDER_SOURCE", "WEB");
        return message;
    }
});
```

Обратите внимание, что `MessagePostProcessor` – это функциональный интерфейс, а значит, его реализацию можно немного упростить, заменив внутренний анонимный класс лямбда-выражением:

```
jms.convertAndSend("tacocloud.order.queue", order,
    message -> {
        message.setStringProperty("X_ORDER_SOURCE", "WEB");
        return message;
});
```

В данном случае этот конкретный экземпляр `MessagePostProcessor` нужен только в одном вызове `convertAndSend()`, но если вдруг обнаружится, что одна и та же реализация `MessagePostProcessor` используется в нескольких разных вызовах `convertAndSend()`, то предпочтительнее использовать ссылку на метод, как показано ниже. Это поможет избежать ненужного дублирования кода:

```
@GetMapping("/convertAndSend/order")
public String convertAndSendOrder() {
    TacoOrder order = buildOrder();
    jms.convertAndSend("tacocloud.order.queue", order,
        this::addOrderSource);
    return "Convert and sent order";
}

private Message addOrderSource(Message message) throws JMSException {
    message.setStringProperty("X_ORDER_SOURCE", "WEB");
    return message;
}
```

Итак, мы рассмотрели несколько способов отправки сообщений. Но какой смысл отправлять сообщения, если их никто никогда не получит? Давайте посмотрим, как организовать получение сообщения с помощью Spring JMS.

9.1.3 Получение сообщений с помощью JMS

Когда дело доходит до получения сообщений, появляется возможность выбора между моделями *активного (pull)*, когда клиентский код запрашивает получение сообщения и ждет его поступления, и *пассивного (push)* извлечения сообщений, когда клиент автоматически получает сообщения по мере их появления.

`JmsTemplate` поддерживает несколько способов получения сообщений, но все они используют активную модель извлечения. Клиент вызывает один из доступных методов, запрашивая получение сообщения, и блокируется, пока сообщение не будет доступно (что может произойти немедленно или спустя какое-то время).

С другой стороны, у нас есть возможность использовать модель пассивного извлечения сообщений. Для этого нужно реализовать приемник сообщений, который будет вызываться каждый раз, когда появится доступное для извлечения сообщение.

Оба варианта подходят для разных случаев использования. Принято считать, что модель пассивного получения (push) – лучший выбор, потому что не блокирует поток выполнения. Но иногда приемник может не справляться с нагрузкой, если сообщения поступают слишком часто. Модель активного извлечения (pull) позволяет получателю объявить, что он готов обработать следующее сообщение.

Давайте рассмотрим оба способа и начнем с модели активного извлечения (pull), которая поддерживается в `JmsTemplate`.

ПОЛУЧЕНИЕ СООБЩЕНИЙ С ПОМОЩЬЮ `JmsTemplate`

`JmsTemplate` предлагает несколько методов извлечения сообщений из брокера, в том числе:

```
Message receive() throws JmsException;
Message receive(Destination destination) throws JmsException;
Message receive(String destinationName) throws JmsException;

Object receiveAndConvert() throws JmsException;
Object receiveAndConvert(Destination destination) throws JmsException;
Object receiveAndConvert(String destinationName) throws JmsException;
```

Как видите, эти шесть методов являются зеркальным отражением методов `send()` и `convertAndSend()` из `JmsTemplate`. Методы `receive()` возвращают сообщение `Message` без всякой обработки, а методы `receiveAndConvert()` используют настроенный конвертер для преобразования сообщений в типы данных предметной области. Также есть возможность передать место назначения в виде экземпляра `Destination` или строки с его именем либо получить сообщение из места назначения по умолчанию.

Чтобы опробовать эти методы, напишем код, извлекающий `TacoOrder` из места назначения `tacocloud.order.queue`. В листинге 9.2 по-

казана реализация интерфейса `OrderReceiver`, получающая экземпляр заказа с помощью `JmsTemplate.receive()`.

Листинг 9.2 Активное извлечение заказов из очереди

```
package tacos.kitchen.messaging.jms;
import javax.jms.Message;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.support.converter.MessageConverter;
import org.springframework.stereotype.Component;

@Component
public class JmsOrderReceiver implements OrderReceiver {

    private JmsTemplate jms;
    private MessageConverter converter;

    @Autowired
    public JmsOrderReceiver(JmsTemplate jms, MessageConverter converter) {
        this.jms = jms;
        this.converter = converter;
    }

    public TacoOrder receiveOrder() {
        Message message = jms.receive("tacocloud.order.queue");
        return (TacoOrder) converter.fromMessage(message);
    }
}
```

Здесь мы указали строку с именем места назначения, откуда должен быть извлечен заказ. Метод `receive()` возвращает экземпляр сообщения `Message`, но нам нужен экземпляр `TacoOrder`, находящийся внутри `Message`, поэтому далее мы используем внедренный конвертер сообщений, чтобы получить экземпляр `TacoOrder`. Действия конвертера направляются свойством объекта `Message`, хранящим идентификатор типа `TacoOrder`, но он возвращает требуемый нам экземпляр как `Object`, поэтому мы дополнительно осуществляем приведение к типу `TacoOrder`.

Получение необработанного объекта `Message` может быть удобно в некоторых случаях, особенно когда требуется проверить свойства и заголовки сообщения. Но чаще бывает нужна только полезная нагрузка. Преобразование этой полезной нагрузки в тип данных предметной области выполняется в два этапа и требует внедрения конвертера сообщений в компонент. В случаях, когда интерес представляет только полезная нагрузка сообщения, проще и удобнее использовать метод `receiveAndConvert()`. В листинге 9.3 показано, как реорганизовать `JmsOrderReceiver` и использовать `receiveAndConvert()` вместо `receive()`.

Листинг 9.3 Получение преобразованного объекта TacoOrder

```
package tacos.kitchen.messaging.jms;

import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;
import tacos.TacoOrder;
import tacos.kitchen.OrderReceiver;

@Component
public class JmsOrderReceiver implements OrderReceiver {

    private JmsTemplate jms;

    public JmsOrderReceiver(JmsTemplate jms) {
        this.jms = jms;
    }

    @Override
    public TacoOrder receiveOrder() {
        return (TacoOrder) jms.receiveAndConvert("tacocloud.order.queue");
    }
}
```

В этой новой версии `JmsOrderReceiver` метод `receiveOrder()` сократился до одной строки. Кроме того, отпала необходимость внедрения `MessageConverter`, потому что все преобразования сообщений будут выполняться за кулисами в методе `receiveAndConvert()`.

Прежде чем двигаться дальше, давайте посмотрим, как использовать `receiveOrder()` в кухонном приложении Taco Cloud. Повар на одной из кухонь Taco Cloud может нажать кнопку или выполнить какое-то другое действие, чтобы показать, что он готов приступить к приготовлению тако. В этот момент будет вызван метод `receiveOrder()`, который вызовет `receive()` или `receiveAndConvert()`. После этого приложение приостановится, пока не появится новое сообщение о заказе. Как только это произойдет, `receiveOrder()` вернет управление, и информация из сообщения будет использована для отображения сведений о заказе, чтобы повар мог приступить к работе. Это кажется естественным выбором для активной модели (`pull`).

Теперь давайте посмотрим, как работает пассивная модель (`push`), основанная на создании приемника JMS.

Объявление приемников сообщений

В отличие от модели активного извлечения, в которой для получения сообщения требуется явно вызвать `receive()` или `receiveAndConvert()`, приемник сообщений является пассивным компонентом, который бездействует, пока не появится новое сообщение.

Чтобы создать приемник сообщений, достаточно просто аннотировать метод компонента аннотацией `@JmsListener`. В листинге 9.4

показан новый компонент `OrderListener`, реализующий пассивную модель приема сообщений.

Листинг 9.4 Компонент `OrderListener`, реализующий пассивную модель получения заказов

```
package tacos.kitchen.messaging.jms.listener;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Profile;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

import tacos.TacoOrder;
import tacos.kitchen.KitchenUI;

@Profile("jms-listener")
@Component
public class OrderListener {

    private KitchenUI ui;

    @Autowired
    public OrderListener(KitchenUI ui) {
        this.ui = ui;
    }

    @JmsListener(destination = "tacocloud.order.queue")
    public void receiveOrder(TacoOrder order) {
        ui.displayOrder(order);
    }
}
```

Метод `receiveOrder()` снабжен аннотацией `@JmsListener`, которая превращает его в приемник сообщений, прослушивающий место назначения `tacocloud.order.queue`. Он не имеет отношения к `JmsTemplate` и не вызывается кодом приложения явно – фреймворк Spring автоматически проверяет поступление сообщений в указанное место назначения и в нужный момент вызывает метод `receiveOrder()`, передавая ему полученный экземпляр `TacoOrder`.

Во многих отношениях аннотация `@JmsListener` похожа на аннотации отображения запросов в Spring MVC, такие как `@GetMapping` или `@PostMapping`. В Spring MVC методы, отмеченные аннотациями отображения запросов, вызываются для обработки запросов с определенными путями. Точно так же методы, отмеченные аннотацией `@JmsListener`, вызываются для обработки сообщений, поступающих в указанное место назначения.

Приемники сообщений часто рекламируются как лучший выбор, потому что они не блокируют приложение и могут быстро обработать несколько сообщений. Однако для приложения Taco Cloud это,

пожалуй, не лучший выбор. Повара являются серьезным узким местом в системе и могут оказаться не в состоянии готовить tanto так же быстро, как поступают заказы. Повар может выполнить заказ наполовину, когда на экране отобразится новый заказ. Пользовательский интерфейс приложения для кухни должен буферизовать заказы по мере их поступления, чтобы не перегружать поваров.

Это не значит, что приемники сообщений – плохой выбор. Наоборот, они идеально подходят для быстрой обработки сообщений. Но когда обработчики должны получать сообщения в своем темпе, то более подходящей выглядит активная модель, предлагаемая JmsTemplate.

Поскольку служба JMS определяется стандартной спецификацией Java и поддерживается многими реализациями брокеров сообщений, она часто используется для организации обмена сообщениями в Java. Но JMS имеет несколько недостатков, не последним из которых является ограничение ее применения только приложениями на Java, потому что это все-таки спецификация Java. Новые механизмы обмена сообщениями, такие как RabbitMQ и Kafka, лишены этого недостатка и доступны для использования в других языках и платформах. Поэтому я предлагаю отложить JMS в сторону и познакомиться с реализацией обмена сообщениями через RabbitMQ.

9.2 RabbitMQ и AMQP

RabbitMQ – пожалуй, самая известная реализация AMQP, предлагающая более мощную стратегию маршрутизации сообщений, чем JMS. В JMS сообщения адресуются именем места назначения, откуда получатель извлекает их, сообщения AMQP адресуются именем обменника и ключом маршрутизации, которые отделены от очереди, просматриваемой получателем. Эта связь между обменником и очередями показана на рис. 9.2.

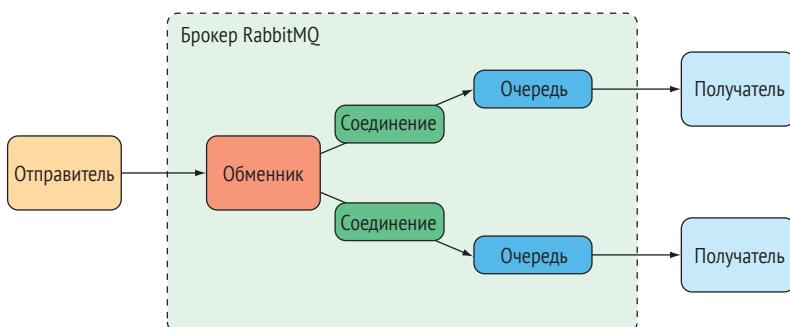


Рис. 9.2 Сообщения, отправляемые в обменник RabbitMQ, направляются в одну или несколько очередей, в зависимости от ключей маршрутизации и соединений

Сообщение, отправленное брокеру RabbitMQ, сохраняется в обменнике (exchange), которому оно было адресовано. Обменник отвечает за маршрутизацию сообщений в одну или несколько очередей, в зависимости от типа обменника, соединений между обменником и очередями и значения ключа маршрутизации сообщения.

Существует несколько видов обменников, в том числе:

- *по умолчанию* – специальный обменник, автоматически создаваемый брокером. Он направляет сообщения в очереди, имена которых совпадают с ключом маршрутизации сообщения. Все очереди автоматически соединяются с обменником по умолчанию;
- *прямой* – направляет сообщения в очередь, ключ соединения которой совпадает с ключом маршрутизации сообщения;
- *тема* – направляет сообщения в одну или несколько очередей с ключом соединения (который может содержать подстановочные знаки), соответствующим ключу маршрутизации сообщения;
- *разветвление* – направляет сообщения во все очереди, соединенные с обменником, без учета ключей соединения или маршрутизации;
- *заголовки* – действует подобно темам, но для маршрутизации использует значения заголовков сообщений, а не ключи маршрутизации;
- *недоставленные сообщения* – собирает все сообщения, которые невозможно доставить (т. е. они не соответствуют какому-то определенному соединению с очередью).

Простейшие формы обменников – обменник по умолчанию и обменник разветвления. Они примерно соответствуют очереди и теме в JMS. Другие обменники позволяют определять более гибкие схемы маршрутизации.

Самое важное, что нужно понять, – сообщения отправляются в обменники с ключами маршрутизации, а извлекаются из очередей. Как они попадают из обменника в очередь, зависит от настроек соединения, лучше всего соответствующих вашим потребностям.

Выбор типа обменника и настроек соединений между обменниками и очередями мало влияет на порядок отправки и приема сообщений в приложениях Spring. Поэтому мы сосредоточимся на том, как писать код, который отправляет и получает сообщения с помощью Rabbit.

ПРИМЕЧАНИЕ Более подробное обсуждение особенностей связывания очередей с обменниками вы найдете в книге Гэвина Роя (Gavin Roy) «RabbitMQ in Depth» (Manning, 2017) или в книге Альваро Видела (Alvaro Videla) и Джейсона Дж. В. Уильямса (Jason J. W. Williams) «RabbitMQ in Action» (Manning, 2012).

9.2.1 Добавление поддержки RabbitMQ в приложение Spring

Прежде чем начать отправлять и получать сообщения в приложении Spring через RabbitMQ, необходимо добавить в спецификацию сборки начальную зависимость Spring Boot AMQP и убрать зависимость Artemis или ActiveMQ, добавленную в предыдущем разделе:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

Добавление начальной зависимости AMQP в спецификацию сборки запустит механизм автоконфигурации, который создаст фабрику соединений AMQP, компоненты `RabbitTemplate` и другие вспомогательные компоненты. Простого добавления этой зависимости вполне достаточно, чтобы получить возможность отправлять и получать сообщения через брокера RabbitMQ с помощью Spring. Однако есть несколько полезных свойств, о которых вам следует знать (табл. 9.4).

В окружении разработки у вас, вероятно, будет использоваться брокер RabbitMQ, не требующий аутентификации, действующий на локальном компьютере и прослушивающий порт 5672. Вам едва ли понадобится изменять эти свойства в окружении разработки, но их, без сомнения, придется настраивать при развертывании приложения в промышленном окружении.

Таблица 9.4 Свойства для настройки местоположения и учетных данных брокера RabbitMQ

Свойство	Описание
<code>spring.rabbitmq.addresses</code>	Список адресов брокера RabbitMQ, перечисленных через запятую
<code>spring.rabbitmq.host</code>	Хост брокера (по умолчанию: <code>localhost</code>)
<code>spring.rabbitmq.port</code>	Порт брокера (по умолчанию: 5672)
<code>spring.rabbitmq.username</code>	Имя пользователя для доступа к брокеру (необязательно)
<code>spring.rabbitmq.password</code>	Пароль для доступа к брокеру (необязательно)

ЗАПУСК БРОКЕРА RABBITMQ Если у вас еще нет действующего брокера RabbitMQ на локальном компьютере, то обращайтесь к официальной документации RabbitMQ по адресу <https://www.rabbitmq.com/download.html>, где вы найдете самые свежие инструкции по запуску RabbitMQ.

Например, предположим, что в вашем промышленном окружении брокер RabbitMQ находится на сервере с именем `rabbit.tacocloud.com`, прослушивает порт 5673 и требует передачи учетных данных. В таком случае все необходимые свойства установит следующая конфигурация в файле `application.yml` при активизации профиля `prod`:

```
spring:
  profiles: prod
  rabbitmq:
    host: rabbit.tacocloud.com
    port: 5673
    username: tacoweb
    password: l3tm31n
```

После настройки RabbitMQ в приложении можно начинать отправлять сообщения с помощью `RabbitTemplate`.

9.2.2 Отправка сообщений с помощью `RabbitTemplate`

Основой поддержки обмена сообщениями через RabbitMQ в Spring является класс `RabbitTemplate`. Он похож на `JmsTemplate` и предлагает аналогичный набор методов. Однако, как вы увидите далее, имеются некоторые тонкие отличия, обусловленные уникальными особенностями RabbitMQ.

Для отправки сообщений `RabbitTemplate` предлагает методы `send()` и `convertAndSend()`, аналогичные методам `JmsTemplate`. Но, в отличие от методов `JmsTemplate`, которые отправляют сообщения только в заданную очередь или тему, методы `RabbitTemplate` отправляют сообщения с учетом заданных обменников и ключей маршрутизации. Вот несколько наиболее простых способов отправки сообщений с помощью `RabbitTemplate`¹:

```
// Отправка неструктурированных сообщений
void send(Message message) throws AmqpException;
void send(String routingKey, Message message) throws AmqpException;
void send(String exchange, String routingKey, Message message)
          throws AmqpException;

// Отправка сообщений в форме объектов
void convertAndSend(Object message) throws AmqpException;
void convertAndSend(String routingKey, Object message)
          throws AmqpException;
void convertAndSend(String exchange, String routingKey,
          Object message) throws AmqpException;

// Отправка сообщений в форме объектов с постобработкой
void convertAndSend(Object message, MessagePostProcessor mPP)
          throws AmqpException;
void convertAndSend(String routingKey, Object message,
          MessagePostProcessor messagePostProcessor)
          throws AmqpException;
void convertAndSend(String exchange, String routingKey,
          Object message,
          MessagePostProcessor messagePostProcessor)
          throws AmqpException;
```

¹ Эти методы определяются интерфейсом `AmqpTemplate`, который реализует класс `RabbitTemplate`.

Как видите, эти методы аналогичны их близнецам в `JmsTemplate`. Все три метода `send()` отправляют необработанный объект `Message`. Следующие три метода `convertAndSend()` принимают прикладной объект, который будет конвертирован в экземпляр `Message` перед отправкой. Последние три метода `convertAndSend()` аналогичны предыдущим трем, но принимают `MessagePostProcessor`, с помощью которого можно внести дополнительные корректировки в объект `Message` перед отправкой брокеру.

В отличие от своих аналогов в `JmsTemplate` вместо имени места назначения (или объекта `Destination`) эти методы принимают строковые значения с именем обменника и ключом маршрутизации. Методы, не имеющие параметра `exchange`, будут отправлять сообщения в обменник по умолчанию. Аналогично методы, не имеющие параметра `routingKey`, будут маршрутизировать сообщения с использованием ключа маршрутизации по умолчанию.

Давайте попробуем реализовать отправку заказов на тако с помощью `RabbitTemplate`. Один из возможных способов – использовать метод `send()`, как показано в листинге 9.5. Но, прежде чем вызвать `send()`, необходимо преобразовать объект `TacoOrder` в `Message`. Это было бы весьма трудоемкой задачей, если бы не простота получения конвертера сообщений с помощью метода `getMessageConverter()`.

Листинг 9.5 Отправка сообщения с помощью `RabbitTemplate.send()`

```
package tacos.messaging;

import org.springframework.amqp.core.Message;
import org.springframework.amqp.core.MessageProperties;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import tacos.Order;

@Service
public class RabbitOrderMessagingService
    implements OrderMessagingService {

    private RabbitTemplate rabbit;

    @Autowired
    public RabbitOrderMessagingService(RabbitTemplate rabbit) {
        this.rabbit = rabbit;
    }

    public void sendOrder(TacoOrder order) {
        MessageConverter converter = rabbit.getMessageConverter();
        MessageProperties props = new MessageProperties();
        Message message = converter.toMessage(order, props);
        rabbit.convertAndSend("tacos-orders", "order", message);
    }
}
```

```

    rabbit.send("tacocloud.order", message);
}

}

```

Обратите внимание, что RabbitOrderMessagingService реализует OrderMessagingService, как и JmsOrderMessagingService, а это значит, что его точно так же можно внедрить в OrderApiController, как для отправки заказа при его размещении. У нас пока нет кода, который получал бы эти сообщения, поэтому используем веб-консоль управления RabbitMQ. Информацию об установке, настройке и использовании консоли RabbitMQ вы найдете на странице <https://www.rabbitmq.com/management.html>.

Имея MessageConverter, конвертировать TacoOrder в Message несложно. Любые свойства сообщения должны передаваться в вид экземпляра MessageProperties, но если сообщение не имеет каких-либо особенных свойств, то можно использовать экземпляр MessageProperties по умолчанию. После этих подготовительных операций остается только вызвать метод send() и передать ему имя обменника и ключ маршрутизации (оба параметра являются необязательными) вместе с сообщением. В этом примере вместе с сообщением мы передали только ключ маршрутизации tacocloud.order, поэтому будет использоваться обменник по умолчанию.

Обменник по умолчанию имеет имя "" (пустая строка) и автоматически создается брокером RabbitMQ. Точно так же ключ маршрутизации по умолчанию является пустой строкой (его маршрутизация зависит от обменника и соединений). Все эти значения по умолчанию можно переопределить, установив свойства spring.rabbitmq.template.exchange и spring.rabbitmq.template.routing-key:

```

spring:
  rabbitmq:
    template:
      exchange: tacocloud.order
      routing-key: kitchens.central

```

В этом случае все сообщения, отправляемые в обменник по умолчанию, будут автоматически отправляться в обменник с именем tacocloud.order. Также если в вызове send() или convertAndSend() не указан ключ маршрутизации, то сообщения будут маршрутизироваться с ключом kitchens.central.

Создать объект Message с помощью конвертера сообщений достаточно просто, но еще проще использовать convertAndSend(), чтобы переложить всю работу на RabbitTemplate:

```

public void sendOrder(TacoOrder order) {
  rabbit.convertAndSend("tacocloud.order", order);
}

```

Настройка конвертера сообщений

По умолчанию преобразование сообщений выполняется с помощью класса `SimpleMessageConverter`, который может преобразовывать в экземпляры `Message` простые типы (например, `String`) и объекты, реализующие интерфейс `Serializable`. Но Spring предлагает для `RabbitTemplate` еще несколько конвертеров сообщений, в том числе:

- `Jackson2JsonMessageConverter` – преобразует объекты в формат JSON и обратно с помощью процессора Jackson 2 JSON;
- `MarshallingMessageConverter` – выполняет преобразования с помощью `Marshaller` и `Unmarshaller` в Spring;
- `SerializerMessageConverter` – преобразует строки и простые объекты любого типа, используя абстракции `Serializer` и `Deserializer`;
- `SimpleMessageConverter` – преобразует строки, массивы байтов и сериализуемые типы;
- `ContentTypeDelegatingMessageConverter` – делегирует выполнение преобразований другому `MessageConverter`, опираясь на заголовок `contentType`;
- `MessagingMessageConverter` – делегирует преобразование сообщений базовому `MessageConverter` и заголовков – `AmqpHeaderConverter`.

Если вам потребуется сменить конвертер сообщений, то просто настройте компонент `MessageConverter`. Например, для преобразования сообщений в формат JSON и обратно можно настроить `Jackson2JsonMessageConverter`:

```
@Bean
public Jackson2JsonMessageConverter messageConverter() {
    return new Jackson2JsonMessageConverter();
}
```

Механизм автоконфигурации в Spring Boot обнаружит этот компонент и внедрит его в `RabbitTemplate` вместо конвертера сообщений по умолчанию.

Настройка свойств сообщения

Так же как при использовании JMS, иногда может потребоваться установить некоторые заголовки в отправляемых сообщениях. Например, представьте, что нам нужно добавить заголовок `X_ORDER_SOURCE` во все сообщения с заказами, полученными через веб-сайт Taco Cloud. Создавая свои объекты `Message`, мы можем установить заголовок с помощью экземпляра `MessageProperties`, который передается конвертеру сообщений. Если вернуться к методу `sendOrder()` в листинге 9.5, то нам понадобится добавить только одну строку кода, как показано ниже:

```
public void sendOrder(TacoOrder order) {
    MessageConverter converter = rabbit.getMessageConverter();
```

```

MessageProperties props = new MessageProperties();
props.setHeader("X_ORDER_SOURCE", "WEB");
Message message = converter.toMessage(order, props);
rabbit.send("tacocloud.order", message);
}

```

Однако при использовании `convertAndSend()` у вас нет доступа к объекту `MessageProperties`. В таких ситуациях может помочь `MessagePostProcessor`:

```

public void sendOrder(TacoOrder order) {
    rabbit.convertAndSend("tacocloud.order.queue", order,
        new MessagePostProcessor() {
            @Override
            public Message postProcessMessage(Message message)
                throws AmqpException {
                MessageProperties props = message.getMessageProperties();
                props.setHeader("X_ORDER_SOURCE", "WEB");
                return message;
            }
        });
}

```

Здесь мы передаем в вызов `convertAndSend()` анонимную реализацию `MessagePostProcessor`. В методе `postProcessMessage()` мы извлекаем `MessageProperties` из сообщения и устанавливаем заголовок `X_ORDER_SOURCE` вызовом `setHeader()`.

Теперь, когда вы узнали, как отправлять сообщения с помощью `RabbitTemplate`, переключим наше внимание на код, который получает сообщения из очереди RabbitMQ.

9.2.3 Получение сообщений из RabbitMQ

Как мы увидели, отправка сообщений с помощью `RabbitTemplate` мало отличается от отправки сообщений с помощью `JmsTemplate`. И как оказывается, получение сообщений из очереди RabbitMQ не сильно отличается от получения сообщений из JMS.

Так же как при использовании JMS, у нас есть два варианта:

- активное извлечение сообщений из очереди с помощью `RabbitTemplate`;
- пассивный прием сообщений в методе с аннотацией `@RabbitListener`.

Начнем с активного извлечения сообщений вызовом метода `RabbitTemplate.receive()`.

Получение сообщений с помощью RabbitTemplate

`RabbitTemplate` имеет несколько методов извлечения сообщений из очереди. Вот некоторые наиболее полезные из них:

```

// Извлечение сообщений
Message receive() throws AmqpException;
Message receive(String queueName) throws AmqpException;
Message receive(long timeoutMillis) throws AmqpException;
Message receive(String queueName, long timeoutMillis) throws AmqpException;

// Извлечение объектов из сообщений
Object receiveAndConvert() throws AmqpException;
Object receiveAndConvert(String queueName) throws AmqpException;
Object receiveAndConvert(long timeoutMillis) throws AmqpException;
Object receiveAndConvert(String queueName, long timeoutMillis)
    throws AmqpException;

// Извлечение объектов из сообщений с соблюдением безопасности типов
<T> T receiveAndConvert(ParameterizedTypeReference<T> type)
    throws AmqpException;
<T> T receiveAndConvert(
    String queueName, ParameterizedTypeReference<T> type)
    throws AmqpException;
<T> T receiveAndConvert(
    long timeoutMillis, ParameterizedTypeReference<T> type)
    throws AmqpException;
<T> T receiveAndConvert(String queueName, long timeoutMillis,
    ParameterizedTypeReference<T> type)
    throws AmqpException;

```

Эти методы являются зеркальным отражением методов `send()` и `convertAndSend()`, описанных выше. Методы `receive()` возвращают сообщение `Message` без всякой обработки, а методы `receiveAndConvert()` используют настроенный конвертер для преобразования сообщений в типы данных предметной области.

Но в сигнатурах методов есть несколько заметных отличий. Во-первых, ни один из этих методов не имеет параметра с именем обменника или ключом маршрутизации. Это связано с тем, что имя обменника и ключ маршрутизации необходимы, чтобы доставить сообщение в очередь, но для извлечения сообщения из очереди эта информация не требуется. Приложениям-потребителям не нужно знать имя обменника или ключ маршрутизации. Очередь – это единственное, о чем должны знать приложения-потребители.

Отметьте также, что многие методы принимают параметр `Long`, определяющий время ожидания сообщений в миллисекундах. По умолчанию время ожидания составляет 0 мс, т. е. вызов `receive()` сразу же вернет управление, даже если в очереди нет сообщений. Это заметное отличие от поведения методов `receive()` в `JmsTemplate`. Задавая значение тайм-аута, можно блокировать методы `receive()` и `receiveAndConvert()` до тех пор, пока не поступит сообщение или пока не истечет тайм-аут. Но, даже задавая ненулевой тайм-аут, вы должны быть готовы к тому, что ваш код вернет `null`.

Давайте посмотрим, как можно использовать эти методы на практике. В листинге 9.6 показана новая реализация `OrderReceiver`

на основе Rabbit, которая использует RabbitTemplate для получения заказов.

Листинг 9.6 Извлечение заказов из RabbitMQ с помощью RabbitTemplate

```
package tacos.kitchen.messaging.rabbit;

import org.springframework.amqp.core.Message;
import org.springframework.amqp.rabbit.core.RabbitTemplate;
import org.springframework.amqp.support.converter.MessageConverter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class RabbitOrderReceiver {

    private RabbitTemplate rabbit;
    private MessageConverter converter;

    @Autowired
    public RabbitOrderReceiver(RabbitTemplate rabbit) {
        this.rabbit = rabbit;
        this.converter = rabbit.getMessageConverter();
    }

    public TacoOrder receiveOrder() {
        Message message = rabbit.receive("tacocloud.order");
        return message != null
            ? (TacoOrder) converter.fromMessage(message)
            : null;
    }
}
```

Все действия происходят в методе `receiveOrder()`. Он вызывает метод `receive()` внедренного компонента RabbitTemplate, чтобы получить заказ из очереди `tacocloud.order`. Значение тайм-аута не определено, поэтому можно предположить, что вызов немедленно вернет сообщение `Message` или значение `null`. Если возвращается сообщение, то с помощью `MessageConverter` из RabbitTemplate оно преобразуется в `TacoOrder`. Если же вызов `receive()` вернет `null`, то и `receiveOrder()` вернет `null`.

В зависимости от потребностей можно допустить небольшую задержку. Например, при выводе информации на табло кухни Taco Cloud можно немного подождать, если нет других заказов. Допустим, мы решили, что можно потратить на ожидание до 30 секунд. Тогда метод `receiveOrder()` можно изменить так, чтобы он передал 30 000-миллисекундную задержку в вызов метода `receive()`:

```
public TacoOrder receiveOrder() {
    Message message = rabbit.receive("tacocloud.order.queue", 30000);
```

```

    return message != null
        ? (TacoOrder) converter.fromMessage(message)
        : null;
}

```

Если у вас, как и у меня, такие жестко закодированные числа вызывают небольшой дискомфорт, то вы могли бы подумать, что неплохо было бы создать класс с аннотацией `@ConfigurationProperties` и с его помощью настроить время ожидания в виде конфигурационного свойства Spring Boot. Я бы с вами согласился, если бы не тот факт, что Spring Boot уже предлагает такое свойство. Если вы решите устанавливать тайм-аут через конфигурацию, то просто удалите значение из вызова `receive()` и установите его в своей конфигурации в виде свойства `spring.rabbitmq.template.receive-timeout`:

```

spring:
  rabbitmq:
    template:
      receive-timeout: 30000

```

Но вернемся к методу `receiveOrder()`: обратите внимание, что нам пришлось использовать конвертер сообщений из `RabbitTemplate` для преобразования объекта `Message` в объект `TacoOrder`. Но если `RabbitTemplate` содержит конвертер сообщений, то почему он не может сделать это преобразование автоматически? Потому что для этой цели предназначен метод `receiveAndConvert()`. Используя метод `receiveAndConvert()`, можно переписать метод `receiveOrder()` так:

```

public TacoOrder receiveOrder() {
    return (TacoOrder) rabbit.receiveAndConvert("tacocloud.order.queue");
}

```

Как видите, он стал намного проще. Единственное, что меня беспокоит, – это приведение типа `Object` к типу `TacoOrder`. Однако есть альтернативное решение: можно передать `ParameterizedTypeReference` в вызов `receiveAndConvert()` и сразу получить объект `TacoOrder`:

```

public TacoOrder receiveOrder() {
    return rabbit.receiveAndConvert("tacocloud.order.queue",
        new ParameterizedTypeReference<Order>() {});
}

```

Насколько этот подход лучше – спорный вопрос, но это, безусловно, более безопасное решение, чем приведение типов. Единственное требование к использованию `ParameterizedTypeReference` с `receiveAndConvert()` – конвертер сообщений должен быть реализацией `SmartMessageConverter`; единственная готовая реализация этого интерфейса – `Jackson2JsonMessageConverter`.

Модель активного извлечения сообщений, предлагаемая `RabbitTemplate`, с успехом может использоваться во многих случаях, но часто лучше иметь код, который пассивно ожидает появления сообщений

и вызывается при их поступлении. Давайте посмотрим, как можно реализовать bean-компонент, который реагирует на появление новых сообщений RabbitMQ.

Пассивный прием сообщений из RabbitMQ

Для создания компонентов на основе RabbitMQ, управляемых сообщениями, Spring предлагает аннотацию `@RabbitListener`, аналог аннотации `@JmsListener`. Эту аннотацию можно применить к методу, который должен вызываться при поступлении сообщения в очередь RabbitMQ.

Например, в листинге 9.7 показана реализация `OrderReceiver` с методом, предназначенным для пассивного приема сообщений о заказах.

Листинг 9.7 Объявление метода для пассивного приема сообщений из очереди RabbitMQ

```
package tacos.kitchen.messaging.rabbit.listener;

import org.springframework.amqp.rabbit.annotation.RabbitListener;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import tacos.TacoOrder;
import tacos.kitchen.KitchenUI;

@Component
public class OrderListener {

    private KitchenUI ui;

    @Autowired
    public OrderListener(KitchenUI ui) {
        this.ui = ui;
    }

    @RabbitListener(queues = "tacocloud.order.queue")
    public void receiveOrder(TacoOrder order) {
        ui.displayOrder(order);
    }
}
```

Этот код очень похож на код из листинга 9.4, не находите? Единственное отличие – это аннотация – `@RabbitListener` вместо `@JmsListener`. Какой бы замечательной ни была аннотация `@RabbitListener`, такое повторение не оставляет мне шанса сказать хоть что-то новое о `@RabbitListener`, чего я не говорил о `@JmsListener`. Они обе отлично подходят для реализации методов, обрабатывающих сообщения, которые им посылают соответствующие брокеры – брокер JMS для `@JmsListener` и брокер RabbitMQ для `@RabbitListener`.

Кто-то может заметить в предыдущем абзаце нехватку энтузиазма по поводу `@RabbitListener`, но, уверяю вас, это несколько ошибочное впечатление. По правде говоря, сам факт, что `@RabbitListener` действует так же, как `@JmsListener`, весьма впечатляет! Это означает, что нам не нужно изучать совершенно другую модель программирования при переходе от Artemis или ActiveMQ к RabbitMQ. То же относится и к сходству между `RabbitTemplate` и `JmsTemplate`.

Давайте сохраним это воодушевление и завершим данную главу знакомством с еще одним механизмом обмена сообщениями, который поддерживается в Spring: Apache Kafka.

9.3 Обмен сообщениями с помощью Kafka

Apache Kafka – самый новый механизм обмена сообщениями из рассматривавшихся в этой главе. На первый взгляд, Kafka – это брокер сообщений, такой же, как ActiveMQ, Artemis или Rabbit. Но у Kafka есть несколько уникальных особенностей.

Kafka предназначен для работы в кластере и обеспечивает отличную масштабируемость. Благодаря распределению своих тем по всем экземплярам в кластере, он очень устойчив. В отличие от брокера RabbitMQ, обслуживающего очереди в обменниках, Kafka использует темы и реализует обмен сообщениями только по схеме публикация/подписка.

Темы Kafka распространяются между всеми брокерами в кластере. Каждый узел выступает в роли лидера для одной или нескольких тем, отвечая за сохранность данных этой темы и пересылая их на другие узлы в кластере.

Кроме того, каждую тему можно разделить на несколько разделов. В этом случае каждый узел в кластере является лидером для одного или нескольких разделов темы, но не для всей темы. Ответственность за тему делится между всеми узлами. На рис. 9.3 показано, как это работает.

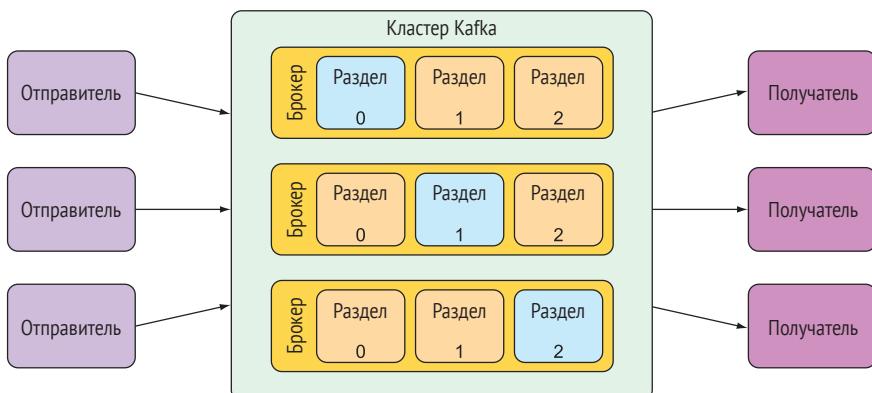


Рис. 9.3 Кластер Kafka состоит из нескольких брокеров, каждый из которых выступает в роли лидера для группы тем

Желающим узнать больше об уникальной архитектуре Kafka я рекомендую прочитать книгу Дилана Скотта (Dylan Scott), Виктора Гамова (Viktor Gamov) и Дейва Клейна (Dave Klein) «Kafka in action» (Manning, 2021). Мы же не будем исследовать устройство Kafka и сосредоточимся исключительно на приемах отправки и получения сообщений с помощью Kafka и Spring.

9.3.1 Настройка Spring для обмена сообщениями через Kafka

Чтобы использовать Kafka для обмена сообщениями, нужно добавить соответствующие зависимости в спецификацию сборки. Однако, в отличие от JMS и RabbitMQ, в Spring Boot нет начальной зависимости для Kafka. Но пусть вас это не беспокоит, потому что вам понадобится только одна зависимость:

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

Эта единственная зависимость добавит в проект все, что нужно для работы с Kafka. Более того, она запустит автоконфигурацию Kafka в Spring Boot, которая, кроме всего прочего, разместит компонент KafkaTemplate в контексте приложения Spring. Вам останется только внедрить его и использовать для отправки и получения сообщений.

Однако, прежде чем начать отправлять и получать сообщения, мы должны познакомиться с некоторыми свойствами, которые пригодятся нам при работе с Kafka. В частности, KafkaTemplate по умолчанию работает с брокером Kafka, который действует на локальном хосте и прослушивает порт 9092. Локальный брокер Kafka отлично подходит для этапа разработки, но когда придет время развернуть приложение в промышленном окружении, вам потребуется настроить другой хост и порт.

УСТАНОВКА КЛАСТЕРА KAFKA Для опробования примеров, которые приводятся далее в этой главе, вам понадобится кластер Kafka. Документация по Kafka, доступная по адресу <https://kafka.apache.org/quickstart>, поможет вам запустить Kafka на локальном компьютере.

Свойство `spring.kafka.bootstrap-servers` задает местоположение одного или нескольких серверов Kafka, используемых для начального подключения к кластеру Kafka. Например, если один из серверов Kafka в кластере находится по адресу `kafka.tacocloud.com` и прослушивает порт 9092, то вы можете настроить его местоположение в YAML так:

```
spring:
  kafka:
```

bootstrap-servers:
- kafka.tacocloud.com:9092

Но обратите внимание, что имя свойства `spring.kafka.bootstrap-servers` имеет множественное число – оно принимает список серверов. То есть вы можете перечислить в этом свойстве несколько серверов Kafka, как показано ниже:

```
spring:
  kafka:
    bootstrap-servers:
      - kafka.tacocloud.com:9092
      - kafka.tacocloud.com:9093
      - kafka.tacocloud.com:9094
```

Эта конфигурация описывает серверы Kafka для начального подключения, действующие на хосте с именем `kafka.tacocloud.com`. Если вы используете локальный кластер Kafka (что часто имеет место во время разработки), то используйте имя хоста `localhost`, как показано ниже:

```
spring:  
  kafka:  
    bootstrap-servers:  
      - localhost:9092
```

После настройки Kafka в проекте можно приступать к реализации отправки и получения сообщений. Начнем с отправки объектов Taco-Order в Kafka с помощью KafkaTemplate.

9.3.2 Отправка сообщений с помощью KafkaTemplate

Компонент KafkaTemplate близко напоминает свои аналоги в JMS и RabbitMQ. В то же время он имеет существенные отличия, которые становятся заметными уже при первом знакомстве с его методами отправки сообщений:

Первое, что бросается в глаза, – отсутствие методов `convertAndSend()`. Это связано с тем, что `KafkaTemplate` типизируется параметрами типов и может напрямую работать с прикладными типами данных. В некотором смысле все методы `send()` выполняют работу `convertAndSend()`.

Также можно заметить, что `send()` и `sendDefault()` имеют параметры, которые сильно отличаются от тех, что использовались с JMS и Rabbit. При отправке сообщений в Kafka можно указать следующие параметры, управляющие отправкой сообщений:

- тема, в которую должно быть отправлено сообщение (требуется для `send()`);
- раздел темы (необязательно);
- ключ для отправки в записи (необязательно);
- отметка времени (необязательно; по умолчанию `System.currentTimeMillis()`);
- полезная нагрузка (собственно сообщение, обязательно).

Тема и полезная нагрузка – два наиболее важных параметра. Разделы и ключи мало влияют на порядок использования `KafkaTemplate`, за исключением передачи дополнительной информации в вызовы `send()` и `sendDefault()`. Далее мы сосредоточимся на отправке полезной нагрузки в заданную тему и не будем беспокоиться о разделах и ключах.

В вызов метода `send()` также можно передать `ProducerRecord` – нечто большее, чем объект, хранящий все предыдущие параметры в одном месте. Также можно отправить объект `Message`, но для этого потребуется преобразовать прикладные объекты в `Message`. Обычно проще использовать какой-нибудь другой метод, чем создавать и отправлять объект `ProducerRecord` или `Message`.

Используя `KafkaTemplate` и его метод `send()`, можно написать реализацию `OrderMessagingService` на основе Kafka. В листинге 9.8 показано, как она выглядит.

Листинг 9.8 Отправка заказа с помощью `KafkaTemplate`

```
package tacos.messaging;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.core.KafkaTemplate;
import org.springframework.stereotype.Service;

import tacos.TacoOrder;

@Service
public class KafkaOrderMessagingService
        implements OrderMessagingService {

    private KafkaTemplate<String, TacoOrder> kafkaTemplate;

    @Autowired
    public KafkaOrderMessagingService(
```

```

        KafkaTemplate<String, TacoOrder> kafkaTemplate) {
    this.kafkaTemplate = kafkaTemplate;
}

@Override
public void sendOrder(TacoOrder order) {
    kafkaTemplate.send("tacocloud.orders.topic", order);
}
}

```

Метод `sendOrder()` этой новой реализации `OrderMessagingService` вызывает `send()` внедренного компонента `KafkaTemplate`, чтобы отправить `TacoOrder` в тему с именем `tacocloud.orders.topic`. За исключением слова «`Kafka`», разбросанного по всему коду, он не сильно отличается от кода, использующего JMS и Rabbit. И так же как другие реализации `OrderMessagingService`, его можно внедрить в `OrderApiController` и использовать для отправки заказов через Kafka, когда заказы размещаются через конечную точку `/api/orders`.

У нас пока нет своего приемника сообщений из Kafka, поэтому используем консоль для просмотра содержимого темы, куда было отправлено сообщение. Для Kafka доступно несколько консолей управления, включая Offset Explorer (<https://www.kafkatoold.com/>) и Confluent Apache Kafka UI (<http://mng.bz/g1P8>).

Если назначить тему по умолчанию, то можно немного упростить метод `sendOrder()`. Чтобы назначить тему по умолчанию, например `tacocloud.orders.topic`, нужно определить значение свойства `spring.kafka.template.default-topic`:

```

spring:
  kafka:
    bootstrap-servers:
      - localhost:9092
    template:
      default-topic: tacocloud.orders.topic

```

Затем в методе `sendOrder()` можно вызвать `sendDefault()` вместо `send()` и опустить параметр темы, как показано ниже:

```

@Override
public void sendOrder(TacoOrder order) {
    kafkaTemplate.sendDefault(order);
}

```

Теперь, когда у нас есть код, отправляющий сообщения, модно перейти к коду, который будет получать эти сообщения от Kafka.

9.3.3 Получение сообщений из Kafka

Помимо уникальных сигнатур методов `send()` и `sendDefault()`, компонент `KafkaTemplate` отличается от `JmsTemplate` и `RabbitTemplate` еще

и тем, что не предлагает никаких методов для активного извлечения сообщений. То есть единственный способ получить сообщения из темы Kafka с помощью Spring – написать свой приемник для пассивного получения сообщений.

Приемники для пассивного получения сообщений из Kafka определяются как методы с аннотацией `@KafkaListener`. Аннотация `@KafkaListener` близко напоминает аннотации `@JmsListener` и `@RabbitListener` и используется почти так же. В листинге 9.9 показано, как может выглядеть метод, получающий заказы из Kafka.

Листинг 9.9 Использование аннотации `@KafkaListener` для реализации приема сообщений из Kafka

```
package tacos.kitchen.messaging.kafka.listener;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.kafka.annotation.KafkaListener;
import org.springframework.stereotype.Component;

import tacos.Order;
import tacos.kitchen.KitchenUI;

@Component
public class OrderListener {

    private KitchenUI ui;

    @Autowired
    public OrderListener(KitchenUI ui) {
        this.ui = ui;
    }

    @KafkaListener(topics="tacocloud.orders.topic")
    public void handle(TacoOrder order) {
        ui.displayOrder(order);
    }
}
```

Метод `handle()` снабжен аннотацией `@KafkaListener`, которая указывает, что этот метод должен вызываться при поступлении сообщения в тему с именем `tacocloud.orders.topic`. Как можно видеть в листинге 9.9, в вызов `handle()` передается только `TacoOrder` (полезная нагрузка). Но если вам понадобятся дополнительные метаданные из сообщения, то можно объявить, что метод принимает объект `ConsumerRecord` или `Message`.

Например, следующая реализация `handle()` принимает `ConsumerRecord`, чтобы получить возможность записать в журнал раздел и отметку времени сообщения:

```
@KafkaListener(topics="tacocloud.orders.topic")
public void handle(
```

```
TacoOrder order, ConsumerRecord<String, TacoOrder> record) {
    log.info("Received from partition {} with timestamp {}", record.partition(), record.timestamp());
    ui.displayOrder(order);
}
```

Точно так же можно организовать получение `Message` вместо `ConsumerRecord` и добиться того же результата:

```
@KafkaListener(topics="tacocloud.orders.topic")
public void handle(Order order, Message<Order> message) {
    MessageHeaders headers = message.getHeaders();
    log.info("Received from partition {} with timestamp {}", headers.get(KafkaHeaders.RECEIVED_PARTITION_ID),
        headers.get(KafkaHeaders.RECEIVED_TIMESTAMP));
    ui.displayOrder(order);
}
```

Стоит отметить, что полезная нагрузка сообщения доступна как `ConsumerRecord.value()` или `Message.getPayload()`. Это означает, что `TacoOrder` можно получить через эти объекты, а не запрашивать его непосредственно как параметр `handle()`.

Итоги

- Асинхронная передача сообщений позволяет отделять друг от друга взаимодействующие приложения, что обеспечивает ослабление связи между ними и увеличение масштабируемости.
- Spring поддерживает асинхронную передачу сообщений с использованием JMS, RabbitMQ и Apache Kafka.
- Приложения могут использовать клиентов на основе шаблонов (`JmsTemplate`, `RabbitTemplate` и `KafkaTemplate`) для отправки сообщений через брокера.
- Приложения-получатели могут извлекать сообщения, используя модель опроса и тех же клиентов на основе шаблонов.
- Так же приложения-получатели могут принимать сообщения, определяя методы-приемники с помощью аннотации `@JmsListener`, `@RabbitListener` или `@KafkaListener`.

10

Интеграция Spring

В этой главе рассматриваются следующие темы:

- обработка данных в режиме реального времени;
- определение потоков интеграции;
- использование определения Spring Integration Java DSL;
- интеграция с электронной почтой, файловыми и другими внешними системами.

Одна из самых больших неприятностей, с которыми мне приходится сталкиваться во время путешествий, – это долгие перелеты и плохое или отсутствующее соединение с интернетом в полете. Я люблю работать, когда нет возможности заняться чем-то другим, в том числе писать статьи или книги. В отсутствие подключения к сети я оказываюсь в малоприятной ситуации, когда мне нужно получить библиотеку или найти описание чего-то в Javadoc и я не могу этого сделать, из-за чего работа останавливается. Со временем я научился брать с собой книги, чтобы читать их в подобных случаях.

Так же как нам нужно подключение к интернету для продуктивной работы, многим приложениям необходимо подключение к внешним системам. Приложению может потребоваться прочитать или отправить электронное письмо, вызвать внешний API или среагировать на информацию, записанную в базу данных. И поскольку данные принимаются из этих внешних систем или записываются в них, приложению может потребоваться так или иначе обработать данные, чтобы преобразовать их в свой формат или из него.

В этой главе вы увидите, как использовать общие шаблоны интеграции с использованием Spring Integration – готовой к использованию реализации многих шаблонов интеграции, которые перечислены в книге Грегора Хопа (Gregor Hohpe) и Бобби Вульфа (Bobby Woolf) «Enterprise Integration Patterns» (Addison-Wesley, 2003)¹. Каждый шаблон реализован как компонент, через который сообщения передают данные в конвейер. Используя конфигурацию Spring, вы можете собрать эти компоненты в конвейер, по которому проходят данные. Давайте начнем с определения простого потока интеграции, который знакомит со многими функциями и характеристиками работы с Spring Integration.

10.1 Объявление простого потока интеграции

Вообще говоря, Spring Integration позволяет создавать потоки интеграции, через которые приложение может получать или отправлять данные в какой-нибудь ресурс, внешний по отношению к приложению. Одним из таких ресурсов является файловая система. Поэтому среди компонентов Spring Integration есть адаптеры каналов для чтения и записи файлов.

Чтобы попробовать Spring Integration в деле, мы создадим поток интеграции, который записывает данные в файловую систему. Для начала добавим Spring Integration в спецификацию сборки нашего проекта. Для Maven необходимо добавить следующие зависимости:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-integration</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-file</artifactId>
</dependency>
```

Первая зависимость – это начальная зависимость Spring Integration в Spring Boot. Она необходима для любых потоков Spring Integration, независимо от того, с чем эти потоки интегрируются. Как и все начальные зависимости Spring Boot, она доступна также в виде флагка в форме Initializr².

Вторая зависимость подключает модуль конечной точки Spring Integration для доступа к файлам. Кроме этого модуля имеется еще более двух десятков модулей конечных точек, используемых для интеграции с внешними системами. Подробнее о модулях конечных точек

¹ Хоп Грегор, Вульф Бобби. Шаблоны интеграции корпоративных приложений. Вильямс (2016), ISBN: 978-5-8459-1946-5, 0-321-20068-3. – Прим. перев.

² <https://start.spring.io/>.

мы поговорим в разделе 10.2.9, а пока просто помните, что модуль конечной точки для доступа к файловой системе поддерживает возможность читать файлы из файловой системы в поток интеграции и/или записывать данные из потока в файловую систему.

Далее нам нужно реализовать возможность отправлять данные в поток интеграции, чтобы их можно было записать в файл. Для этого создадим интерфейс шлюза (листинг 10.1).

Листинг 10.1 Интерфейс шлюза сообщений для преобразования вызовов методов в сообщения

```
package sia6;

import org.springframework.integration.annotation.MessagingGateway;
import org.springframework.integration.file.FileHeaders;
import org.springframework.messaging.handler.annotation.Header;

@MessagingGateway(defaultRequestChannel="textInChannel") ← Объявление шлюза сообщений
public interface FileWriterGateway {
    void writeToFile(
        @Header(FileHeaders.FILENAME) String filename, String data); ← Выполняет запись в файл
}
```

`FileWriterGateway` – это самый обычный интерфейс Java, но о нем многое можно рассказать. Прежде всего обратите внимание на аннотацию `@MessagingGateway`. Она требует от фреймворка Spring Integration создать реализацию этого интерфейса во время выполнения, подобно тому, как Spring Data автоматически генерирует реализации интерфейсов репозиториев. Эта реализация будет использоваться другими частями кода, когда им потребуется выполнить запись в файл.

Атрибут `defaultRequestChannel` аннотации `@MessagingGateway` определяет канал, куда должны отправляться любые сообщения, возникающие в результате вызова методов интерфейса. В данном случае все сообщения, возникающие в результате вызова `writeToFile()`, будут направляться в канал с `textInChannel`.

Метод `writeToFile()` принимает строковое имя файла и еще одну строку с текстом для записи в файл. Интересно отметить, что параметр `filename` с именем файла снабжен аннотацией `@Header`. В этом случае аннотация `@Header` сообщает, что значение `filename` должно быть помещено в заголовок сообщения, определяемый константой `FileHeaders.FILENAME` в классе `FileHeaders`, которая имеет значение `"file_name"`, а не в тело сообщения. С другой стороны, значение параметра `data` передается в теле сообщения.

Теперь, после создания шлюза сообщений, нам нужно настроить поток интеграции. Несмотря на то что начальная зависимость Spring Integration, которую мы добавили в сборку, обеспечивает необходимую автоконфигурацию Spring Integration, мы все равно должны

определить дополнительные настройки для потоков, отвечающих потребностям приложения. Поддерживаются три варианта конфигураций для объявления потоков интеграции:

- конфигурация XML;
- конфигурация Java;
- конфигурация Java с DSL.

Мы рассмотрим все три варианта и начнем с самого старого – конфигурации XML.

10.1.1 Определение потоков интеграции в XML

В этой книге я старался по мере возможности не использовать конфигурацию в формате XML, однако Spring Integration имеет долгую историю определения потоков интеграции в XML. Поэтому я думаю, что стоит показать хотя бы один пример определения потока интеграции в XML. В листинге 10.2 показано, как настроить поток в XML.

Листинг 10.2 Определение потока интеграции в 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"
       xmlns:int-file="http://www.springframework.org/schema/integration/file"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/spring-beans.xsd
                           http://www.springframework.org/schema/integration
                           http://www.springframework.org/schema/integration/spring-integration.xsd
                           http://www.springframework.org/schema/integration/file
                           http://www.springframework.org/schema/integration/file/spring-
                           integration-file.xsd">

    <int:channel id="textInChannel" /> ←————— Объявление textInChannel

    <int:transformer id="upperCase"
                     input-channel="textInChannel"
                     output-channel="fileWriterChannel"
                     expression="payload.toUpperCase()" /> ←————— Настройка
                                                               преобразования
                                                               текста

    <int:channel id="fileWriterChannel" /> ←————— Объявление fileWriterChannel

    <int-file:outbound-channel-adapter id="writer"
                                         channel="fileWriterChannel"
                                         directory="/tmp/sia6/files"
                                         mode="APPEND"
                                         append-new-line="true" /> ←————— Настройка записи текста в файл

</beans>
```

Рассмотрим подробнее настройки в листинге 10.2:

- первым объявляется канал `textInChannel`. Это тот же канал, который выбран в качестве канала запроса для `FileWriterGateway`. Когда вызывается метод `writeToFile()` экземпляра `FileWriterGateway`, генерированное сообщение публикуется в этом канале;
- затем настраивается преобразователь, который получает сообщения из `textInChannel`. Он использует выражение на языке Spring Expression Language (SpEL) для применения `toUpperCase()` к телу сообщения, после чего результат публикуется в `fileWriterChannel`;
- далее настраивается канал `fileWriterChannel`. Он соединяет преобразователь с адаптером канала вывода;
- наконец, настраивается адаптер канала вывода с использованием пространства имен `int-file`. Это пространство имен XML поддерживается модулем Spring Integration, предназначенным для записи в файлы. Согласно приведенным настройкам, он будет получать сообщения из `fileWriterChannel` и записывать их полезную нагрузку в файл, имя которого указано в заголовке `"file_name"` сообщения, находящемся в каталоге, который определяется атрибутом `directory`. Если файл уже существует, новые сообщения будут добавляться в конец.

Получившийся поток интеграции изображен на рис. 10.1 с использованием графических элементов, стилизованных под шаблоны *Enterprise Integration Patterns*.

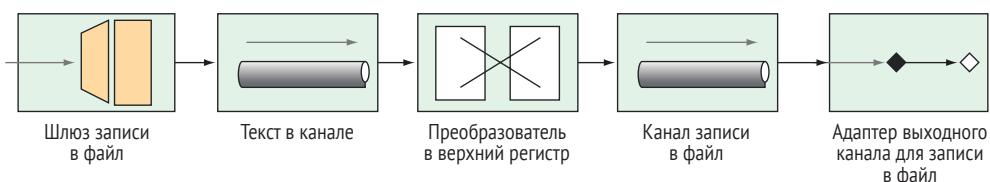


Рис. 10.1 Поток интеграции, выполняющий запись в файл

Поток состоит из пяти компонентов: шлюза, двух каналов, преобразователя и адаптера канала. Это все, что необходимо для получения потока интеграции. Эти и другие компоненты, поддерживаемые Spring Integration, мы рассмотрим в разделе 10.2.

Чтобы использовать конфигурацию XML в приложении Spring Boot, необходимо импортировать XML в качестве ресурса. Самый простой способ сделать это – использовать аннотацию `@ImportResource`, как показано в следующем примере, в одном из конфигурационных Java-классов в приложении:

```

@Configuration
@ImportResource("classpath:/filewriter-config.xml")
public class FileWriterIntegrationConfig { ... }

```

Конфигурация на основе XML долгие годы исправно служила Spring Integration, но многие разработчики стали избегать XML. (И как было

отмечено выше, я тоже стараюсь не использовать XML в этой книге.) Давайте отложим в сторону все эти угловые скобки и рассмотрим конфигурацию Spring Integration в стиле Java.

10.1.2 Определение потоков интеграции в коде на Java

Большинство разработчиков современных приложений Spring отказались от использования конфигурации XML в пользу конфигурации на Java. Фактически в приложениях Spring Boot конфигурация на Java стала естественным стилем, дополняющим автоконфигурацию. Поэтому если в приложение Spring Boot понадобится добавить поток интеграции, то имеет смысл определить его на Java.

В качестве примера такой конфигурации на Java в листинге 10.3 приводится определение того же самого потока записи в файл, что и выше.

Листинг 10.3 Определение потока интеграции на Java

```
package sia6;

import java.io.File;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.integration.annotation.ServiceActivator;
import org.springframework.integration.annotation.Transformer;
import org.springframework.integration.file.FileWriterMessageHandler;
import org.springframework.integration.file.support.FileExistsMode;
import org.springframework.integration.transformer.GenericTransformer;

@Configuration
public class FileWriterIntegrationConfig {

    @Bean
    @Transformer(inputChannel="textInChannel", ← Объявление преобразователя
                 outputChannel="fileWriterChannel")

    public GenericTransformer<String, String> upperCaseTransformer() {
        return text -> text.toUpperCase();
    }

    @Bean
    @ServiceActivator(inputChannel="fileWriterChannel") ← Объявление обработчика
    public FileWriterMessageHandler fileWriter() { ← записи в файл
        FileWritingMessageHandler handler =
            new FileWritingMessageHandler(new File("/tmp/sia6/files"));
        handler.setExpectReply(false);
        handler.setFileExistsMode(FileExistsMode.APPEND);
        handler.setAppendNewLine(true);
        return handler;
    }

}
```

В конфигурации на Java мы объявляем два bean-компоненты: преобразователь и обработчик сообщений для записи в файл. Роль преобразователя играет `GenericTransformer`. Поскольку `GenericTransformer` – это функциональный интерфейс, его реализацию можно оформить в виде лямбда-выражения, вызывающего `toUpperCase()` для преобразования текста сообщения в верхний регистр. Компонент преобразователя снабжен аннотацией `@Transformer`, превращающей его в преобразователь в потоке интеграции, который получает сообщения из входного канала `textInChannel` и записывает результат преобразования в выходной канал `fileWriterChannel`.

Компонент, осуществляющий запись в файлы, снабжен аннотацией `@ServiceActivator`, которая указывает, что сообщения будут извлекаться из `fileWriterChannel` и передаваться службе, представленной экземпляром `FileWritingMessageHandler`, где `FileWritingMessageHandler` – это обработчик сообщений, записывающий полезную нагрузку сообщения в файл с именем в заголовке `"file_name"` сообщения в указанном каталоге. Так же как в примере с XML-конфигурацией, `FileWritingMessageHandler` настроен на добавление новых строк в конец файла.

Одна из уникальных особенностей реализации компонента `FileWritingMessageHandler` – вызов `setExpectReply(false)`, сообщающий, что активатор службы не должен ожидать наличия канала ответа (через который значение можно вернуть компонентам, стоящим выше в потоке). Без вызова `setExpectReply(false)` компонент записи в файл получит значение по умолчанию `true`, и, хотя конвейер продолжит работать как должно, в журнале появятся сообщения об ошибках, указывающие, что канал ответа не был настроен.

Также отметьте, что нет необходимости явно объявлять каналы `TextInChannel` и `fileWriterChannel` – они будут созданы автоматически, если компоненты с такими именами не существуют. Но если понадобится организовать более точное управление настройками каналов, то можно явно сконструировать их как компоненты:

```
@Bean  
public MessageChannel textInChannel() {  
    return new DirectChannel();  
}  
...  
  
@Bean  
public MessageChannel fileWriterChannel() {  
    return new DirectChannel();  
}
```

Вариант реализации конфигурации на Java немного короче и проще читается, и, конечно же, согласуется с подходом оформления конфигурации только на Java, которому я стараюсь следовать в этой книге. Но конфигурацию можно упорядочить еще больше, прибегнув к стилю Java DSL, широко используемому в Spring Integration.

10.1.3 Конфигурация на Spring Integration DSL

Давайте еще раз попытаемся определить поток интеграции для записи в файлы. Эта конфигурация все так же будет определять на Java, но на этот раз на Spring Integration Java DSL. Вместо объявления отдельного bean-компонентента для каждого элемента потока объявим один bean-компонентент, определяющий весь поток, как показано в листинге 10.4.

Листинг 10.4 Использование цепочечного (fluent) API для описания потока интеграции

```
package sia6;

import java.io.File;
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.MessageChannels;
import org.springframework.integration.file.dsl.Files;
import org.springframework.integration.file.support.FileExistsMode;

@Configuration
public class FileWriterIntegrationConfig {

    @Bean
    public IntegrationFlow fileWriterFlow() {
        return IntegrationFlows
            .from(MessageChannels.direct("textInChannel")) ← Входной канал
            .<String, String>transform(t -> t.toUpperCase()) ← Объявление
            .handle(Files ← преобразователя
                .outboundAdapter(new File("/tmp/sia6/files"))
                .fileExistsMode(FileExistsMode.APPEND)
                .appendNewLine(true))
            .get();
    }
}
```

Новая конфигурация лаконична настолько, насколько это возможно, и определяет весь поток в одном методе. Класс `IntegrationFlows` инициирует API построителя, с помощью которого определяет поток.

Поток, представленный в листинге 10.4, начинается с получения сообщений из канала `textInChannel`. Затем сообщения передаются преобразователю в верхний регистр. После преобразователя сообщения обрабатываются адаптером выходного канала, созданным на основе типа `Files`, определяемого в модуле `file` в Spring Integration. Наконец, вызов `get()` создает возвращаемый поток `IntegrationFlow`. Проще говоря, этот метод определяет тот же поток интеграции, что и примеры конфигурации XML и Java.

Обратите внимание, что, как и в примере с конфигурацией на Java, здесь не пришлось явно объявлять компоненты канала, несмотря на наличие ссылки на `textInChannel`. Этот канал будет создан фреймворком Spring Integration автоматически, потому что нигде в коде он не создается явно. Но при желании вы можете создать свой компонент канала.

Канал, соединяющий преобразователь с адаптером выходного канала, мы вообще нигде не используем. Но если потребуется явно настроить его, то можно сослаться на него по имени в определении потока с помощью вызова `channel()`:

```
@Bean
public IntegrationFlow fileWriterFlow() {
    return IntegrationFlows
        .from(MessageChannels.direct("textInChannel"))
        .<String, String>transform(t -> t.toUpperCase())
        .channel(MessageChannels.direct("FileWriterChannel"))
        .handle(Files
            .outboundAdapter(new File("/tmp/sia6/files"))
            .fileExistsMode(FileExistsMode.APPEND)
            .appendNewLine(true))
        .get();
}
```

При применении Spring Integration Java DSL (как и любого другого цепочечного (fluent) API) старайтесь разумно использовать пробелы и отступы для придания удобочитаемого вида. В примере, приведенном выше, я добавил дополнительные отступы в строках, чтобы обозначить блоки взаимосвязанного кода. В еще более длинных и сложных определениях можно даже выделить части потока в отдельные методы или подпотоки, чтобы еще больше улучшить читабельность.

Теперь, когда вы увидели, как определить простой поток с использованием трех разных стилей, отступим на шаг назад и взглянем на общую картину Spring Integration.

10.2 Обзор ландшафта Spring Integration

Spring Integration охватывает множество сценариев интеграции. Попытку включить все это в одну главу можно сравнить с попыткой упаковать слона в почтовый конверт. Поэтому вместо всестороннего обзора Spring Integration я покажу лишь фотографию слона Spring Integration, чтобы вы могли получить хоть какое-то представление о том, как он работает. Затем мы создадим еще один поток интеграции, который пополнит копилку возможностей приложения Taco Cloud.

Типичный поток интеграции состоит из одного или нескольких следующих компонентов. Прежде чем начать писать код, мы кратко рассмотрим роль каждого из них в процессе интеграции:

- **канал** – передает сообщения между компонентами потока;

- *фильтр* – разрешает или запрещает прохождение сообщений через поток, опираясь на некоторые критерии;
- *преобразователь* – изменяет сообщения и/или преобразует полезные данные из одного типа в другой;
- *маршрутизатор* – направляет сообщения в один из нескольких каналов, обычно опираясь на заголовки сообщений;
- *сплиттер (разделитель)* – делит входящие сообщения на два или более сообщений, каждое из которых отправляется в свой канал;
- *агрегатор* – противоположность сплиттеру; объединяет несколько сообщений из разных каналов;
- *активатор службы* – передает сообщение некоторому методу Java для обработки, а затем публикует возвращаемое значение в выходном канале;
- *адаптер канала* – соединяет канал с некоторой внешней системой или транспортом; может осуществлять ввод из внешней системы или вывод в нее;
- *шлюз* – передает данные в поток интеграции через интерфейс.

Вы уже видели некоторые из этих компонентов, когда мы определяли поток интеграции для записи в файлы. Интерфейс `FileWriterGateway` играл роль шлюза, через который приложение отправляло текст в файл. Мы также определили преобразователь для преобразования текста сообщения в верхний регистр. Еще мы объявили шлюз, решающий задачу записи текста в файл. А кроме того, в потоке имелось два канала, `textInChannel` и `fileWriterChannel`, соединявших другие компоненты друг с другом. А теперь перейдем к краткому обзору компонентов потока интеграции.

10.2.1 Каналы сообщений

Каналы сообщений – это средства, с помощью которых сообщения перемещаются по конвейеру интеграции, как показано на рис. 10.2. Каналы соединяют вместе все остальные части Spring Integration.

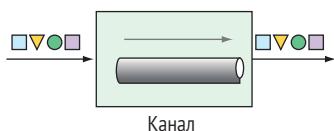


Рис. 10.2 Каналы сообщений – это проводники, по которым данные передаются между компонентами в потоке интеграции

Spring Integration предоставляет несколько реализаций каналов, в том числе:

- *PublishSubscribeChannel* – сообщения, опубликованные в `PublishSubscribeChannel`, передаются одному или нескольким получателям. Если получателей несколько, то все они получат сообщение;
- *QueueChannel* – сообщения, опубликованные в `QueueChannel`, хранятся в очереди, пока не будут извлечены получателем в порядке-

ке их поступления (FIFO). Если получателей несколько, то только один из них получит сообщение;

- *PriorityChannel* – действует аналогично QueueChannel, но сообщения извлекаются получателями не в порядке поступления, а с учетом приоритетов, указанных в заголовках `priority` сообщений;
- *RendezvousChannel* – действует аналогично QueueChannel, за исключением того, что попытка отправителя послать следующее сообщение блокируется, пока получатель не извлечет предыдущее сообщение, что позволяет эффективно синхронизировать работу отправителя и получателя;
- *DirectChannel* – действует аналогично PublishSubscribeChannel, но отправляет сообщение одному получателю, вызывая его в том же потоке выполнения, в котором действует отправитель. Это позволяет передавать транзакции через весь канал;
- *ExecutorChannel* – действует аналогично DirectChannel, но отправка сообщения происходит через TaskExecutor в отдельном потоке выполнения. Каналы этого типа не поддерживают транзакции, охватывающие каналы целиком;
- *FluxMessageChannel* – канал сообщений Reactive Streams Publisher на основе Flux Project Reactor. (Подробнее о Reactive Streams, Reactor и Flux мы поговорим в главе 11.)

В обоих стилях конфигурации – Java и Java DSL – входные каналы создаются автоматически и по умолчанию используется DirectChannel. Но если вы решите применять другую реализацию, вам потребуется явно объявить канал как bean-компонент и сослаться на него в потоке интеграции. Например, чтобы использовать канал PublishSubscribeChannel, нужно объявить следующий метод с аннотацией @Bean:

```
@Bean  
public MessageChannel orderChannel() {  
    return new PublishSubscribeChannel();  
}
```

Затем следует сослаться на этот канал по имени в определении потока интеграции. Например, если канал используется компонентом активатора службы, то сослаться на него можно в атрибуте `inputChannel` аннотации `@ServiceActivator`:

```
@ServiceActivator(inputChannel="orderChannel")
```

Или если используется конфигурация в стиле Java DSL, то сослаться на него можно в вызове `channel()`:

```
@Bean  
public IntegrationFlow orderFlow() {  
    return IntegrationFlows  
        ...
```

```
.channel("orderChannel")
...
.get();
}
```

Следует отметить, что при использовании QueueChannel получатели должны быть настроены на опрос очереди. Например, представьте, что мы объявили компонент QueueChannel следующим образом:

```
@Bean
public MessageChannel orderChannel() {
    return new QueueChannel();
}
```

Тогда мы должны также гарантировать, что получатель будет периодически опрашивать канал на наличие сообщений. В случае активатора службы аннотация @ServiceActivator может выглядеть так:

```
@ServiceActivator(inputChannel="orderChannel",
    poller=@Poller(fixedRate="1000"))
```

В этом примере активатор службы опрашивает канал orderChannel раз в 1 секунду (т. е. раз в 1000 мс).

10.2.2 Фильтры

Вы можете разместить фильтры в середине конвейера интеграции, чтобы управлять передачей сообщений на следующий этап в потоке, как показано на рис. 10.3.

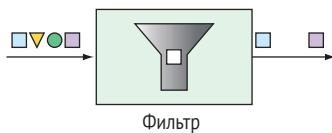


Рис. 10.3 Фильтры основываются на некоторых критериях и разрешают или запрещают прохождение сообщений через конвейер

Например, предположим, что сообщения с целочисленными значениями публикуются через канал numberChannel, но нам нужно, чтобы в канал evenNumberChannel передавались только четные числа. В этом случае можно объявить фильтр с аннотацией @Filter:

```
@Filter(inputChannel="numberChannel",
    outputChannel="evenNumberChannel")
public boolean evenNumberFilter(Integer number) {
    return number % 2 == 0;
}
```

Также, используя стиль конфигурации Java DSL, можно определить поток интеграции, содержащий вызов filter():

```
@Bean
public IntegrationFlow evenNumberFlow(AtomicInteger integerSource) {
```

```

    return IntegrationFlows
        ...
        .<Integer>filter((p) -> p % 2 == 0)
        ...
        .get();
}

```

В этом случае мы реализовали фильтр с использованием лямбда-выражения. Но на самом деле метод `filter()` принимает аргумент типа `GenericSelector`. Это означает, что для организации более сложного критерия фильтрации можно передать реализацию интерфейса `GenericSelector`.

10.2.3 Преобразователи

Преобразователи выполняют некоторую операцию с сообщениями, создавая, по сути, новые сообщения, иногда изменяя тип полезной нагрузки (рис. 10.4). Преобразование может быть простым, как, например, выполнение математических операций с числом или манипулирование строкой, или более сложным, как, например, поиск и возврат сведений о книге по строковому значению ISBN.

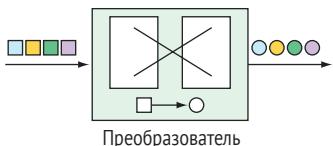


Рис. 10.4 Преобразователи выполняют операции с сообщениями, передаваемыми через поток интеграции

Например, предположим, что в канал `numberChannel` передаются целочисленные значения и их нужно преобразовать в строки, содержащие те же числа, но в римской записи. Для этого можно определить bean-компонент типа `GenericTransformer` и снабдить его аннотацией `@Transformer`:

```

@Bean
@Transformer(inputChannel="numberChannel",
             outputChannel="romanNumberChannel")
public GenericTransformer<Integer, String> romanNumTransformer() {
    return RomanNumbers::toRoman;
}

```

Аннотация `@Transformer` определяет компонент как преобразователь, который получает значения `Integer` из канала `numberChannel` и использует статический метод `toRoman()` для их преобразования. (Метод `toRoman()` определен в классе `RomanNumbers`, и здесь используется ссылка на него.) Результат публикуется в канале `romanNumberChannel`.

В стиле конфигурации Java DSL реализовать преобразование еще проще – достаточно вызвать метод `transform()` и передать ему ссылку на метод `toRoman()`:

```

@Bean
public IntegrationFlow transformerFlow() {
    return IntegrationFlows
        ...
        .transform(RomanNumbers::toRoman)
        ...
        .get();
}

```

В обоих примерах мы использовали ссылку на метод преобразователя, но имейте в виду, что преобразователь также можно реализовать как лямбда-выражение. Или, если преобразователь достаточно сложен, его можно реализовать и внедрить в конфигурацию потока как bean-компонент и передать ссылку в вызов метода `transform()`:

```

@Bean
public RomanNumberTransformer romanNumberTransformer() {
    return new RomanNumberTransformer();
}

@Bean
public IntegrationFlow transformerFlow(
    RomanNumberTransformer romanNumberTransformer) {
    return IntegrationFlows
        ...
        .transform(romanNumberTransformer)
        ...
        .get();
}

```

Здесь мы объявили компонент типа `RomanNumberTransformer`, который является реализацией интерфейсов `Transformer` или `GenericTransformer`, объявленных в Spring Integration. Компонент внедряется в метод `transformFlow()` и передается в вызов метода `transform()` внутри определения потока интеграции.

10.2.4 Маршрутизаторы

Маршрутизаторы, основанные на некоторых критериях маршрутизации, позволяют организовать ветвления в потоке интеграции и направлять сообщения в разные каналы (рис. 10.5).

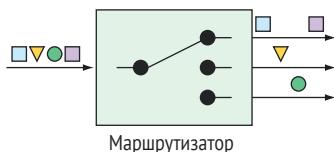


Рис. 10.5 Маршрутизаторы направляют сообщения в разные каналы, основываясь на определенных критериях маршрутизации

Например, предположим, что у нас есть канал `numberChannel`, по которому передаются целочисленные значения. Допустим также, что

нам нужно направлять все сообщения с четными номерами в канал evenChannel, а сообщения с нечетными номерами – в канал oddChannel. Чтобы организовать такую маршрутизацию в потоке интеграции, можно объявить компонент типа AbstractMessageRouter и снабдить его аннотацией @Router:

```

@Bean
@Router(inputChannel="numberChannel")
public AbstractMessageRouter evenOddRouter() {
    return new AbstractMessageRouter() {
        @Override
        protected Collection<MessageChannel>
            determineTargetChannels(Message<?> message) {
            Integer number = (Integer) message.getPayload();
            if (number % 2 == 0) {
                return Collections.singleton(evenChannel());
            }
            return Collections.singleton(oddChannel());
        }
    };
}

@Bean
public MessageChannel evenChannel() {
    return new DirectChannel();
}

@Bean
public MessageChannel oddChannel() {
    return new DirectChannel();
}

```

Объявленный здесь компонент AbstractMessageRouter принимает сообщения из входного канала numberChannel. Реализация в виде внутреннего анонимного класса проверяет содержимое сообщения и, если это четное число, возвращает канал evenChannel (объявленный как компонент после компонента router). В противном случае число должно быть нечетным и возвращается канал oddChannel (также объявленный в методе объявления компонента).

В конфигурации Java DSL маршрутизаторы создаются вызовом route() в определении потока, как показано ниже:

```

@Bean
public IntegrationFlow numberRoutingFlow(AtomicInteger source) {
    return IntegrationFlows
        ...
        .<Integer, String>route(n -> n%2==0 ? "EVEN":"ODD", mapping -> mapping
            .subFlowMapping("EVEN", sf -> sf
                .<Integer, Integer>transform(n -> n * 10)
                    .handle((i,h) -> { ... })
            )
            .subFlowMapping("ODD", sf -> sf

```

```

        .transform(RomanNumbers::toRoman)
        .handle((i,h) -> { ... })
    )
)
.get();
}

```

Мы можем объявить `AbstractMessageRouter` и передать его в `route()`, но в этом конкретном примере используется лямбда-выражение, определяющее четность или нечетность числа в сообщении. Если число четное, возвращается строковое значение `EVEN`. Если нечетное, то возвращается `ODD`. Затем эти значения используются для выбора маршрута сообщения.

10.2.5 Сплиттеры

Иногда в потоке интеграции бывает нужно разделить сообщение на несколько сообщений, чтобы обработать их независимо. Для этой цели используются сплиттеры (разделители), как показано на рис. 10.6, которые делят и обрабатывают такие разделенные сообщения.

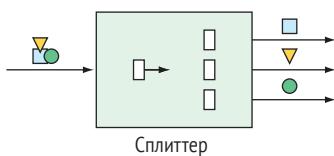


Рис. 10.6 Сплиттеры делят сообщения на две или более частей, которые могут обрабатываться отдельными подпотоками

Сплиттеры могут пригодиться во многих ситуациях, но обычно применяются в двух основных случаях:

- *сообщение содержит коллекцию элементов одного типа, которые можно обрабатывать по отдельности.* Например, сообщение, содержащее список товаров, можно разделить на несколько сообщений, по одному на каждый товар;
- *сообщение содержит информацию, пусть и связанную, но которую можно разделить на два или более сообщений разных типов.* Например, заказ на покупку может содержать адрес доставки, номер счета для оплаты и позиции заказа. Адрес доставки может обрабатываться одним подпотоком, номер счета – другим, а отдельные позиции – третьим. В этом случае за сплиттером обычно следует маршрутизатор, направляющий сообщения по типам хранящихся в них данных, чтобы гарантировать обработку каждой порции данных правильным подпотоком.

При делении содержимого сообщения на два или более сообщений разных типов обычно достаточно определить простой Java-объект (POJO), извлекающий отдельные элементы из содержимого входящего сообщения и возвращающий их как элементы коллекции.

Например, предположим, что нам нужно разделить сообщение с заказом на покупку на два сообщения: одно с платежной информацией,

а другое со списком позиций. Следующий компонент `OrderSplitter` выполнит эту работу:

```
public class OrderSplitter {
    public Collection<Object> splitOrderIntoParts(PurchaseOrder po) {
        ArrayList<Object> parts = new ArrayList<>();
        parts.add(po.getBillingInfo());
        parts.add(po.getLineItems());
        return parts;
    }
}
```

Компонент `OrderSplitter` можно объявить как часть потока интеграции, снабдив его аннотацией `@Splitter`:

```
@Bean
@Splitter(inputChannel="poChannel",
          outputChannel="splitOrderChannel")
public OrderSplitter orderSplitter() {
    return new OrderSplitter();
}
```

Здесь заказы на покупку поступают в канал `poChannel` и делятся с помощью `OrderSplitter`. Затем каждый элемент в полученной коллекции публикуется как отдельное сообщение в канал `splitOrderChannel`. На этом этапе потока можно объявить `PayloadTypeRouter` для маршрутизации информации о выставлении счетов и позиций в отдельные подпотоки:

```
@Bean
@Router(inputChannel="splitOrderChannel")
public MessageRouter splitOrderRouter() {
    PayloadTypeRouter router = new PayloadTypeRouter();
    router.setChannelMapping(
        BillingInfo.class.getName(), "billingInfoChannel");
    router.setChannelMapping(
        List.class.getName(), "lineItemsChannel");
    return router;
}
```

Маршрутизатор `PayloadTypeRouter`, как следует из имени типа, направляет сообщения по разным каналам в зависимости от типа полезной нагрузки. В данном случае сообщения с содержимым типа `BillingInfo` направляются в канал `billingInfoChannel`. Позиции заказа помещаются в коллекцию `java.util.List`, которая соответствует типу `List` и поэтому направляется в канал `lineItemsChannel`.

В этом примере поток делится на два подпотока: в первый передаются объекты `BillingInfo`, а во второй – коллекция `List<LineItem>`. А можно ли затем эту коллекцию разделить на отдельные элементы, чтобы вместо списка объектов `LineItem` обрабатывать объекты `LineItem` по отдельности? Да, можно! Для этого нужно просто разделить

список элементов на отдельные сообщения, написав метод (не компонент) с аннотацией `@Splitter`, который возвращает набор объектов `LineItem`, например так:

```
@Splitter(inputChannel="lineItemsChannel", outputChannel="lineItemChannel")
public List<LineItem> lineItemSplitter(List<LineItem> lineItems) {
    return lineItems;
}
```

Когда сообщение с содержимым `List<LineItem>` поступает в канал `lineItemsChannel`, оно передается в метод `lineItemSplitter()`. Согласно правилам реализации сплиттеров, метод должен возвращать коллекцию элементов, которые необходимо разделить. В этом случае у нас уже есть коллекция объектов `LineItem`, поэтому мы просто возвращаем ее, не выполняя никаких операций с ней. В результате каждый `LineItem` в коллекции будет передан в виде отдельного сообщения в канал `lineItemChannel`.

Если вы предпочитаете использовать Java DSL, то ту же конфигурацию сплиттеров/маршрутизаторов можно получить с помощью вызовов `split()` и `route()`:

```
return IntegrationFlows
    ...
    .split(orderSplitter())
    .<Object, String> route(
        p -> {
            if (p.getClass().isAssignableFrom(BillingInfo.class)) {
                return "BILLING_INFO";
            } else {
                return "LINE_ITEMS";
            }
        }, mapping -> mapping
        .subFlowMapping("BILLING_INFO", sf -> sf
            .<BillingInfo> handle((billingInfo, h) -> {
                ...
            }))
        .subFlowMapping("LINE_ITEMS", sf -> sf
            .split()
            .<LineItem> handle((lineItem, h) -> {
                ...
            }))
    )
    .get();
```

Определение потока в форме DSL, безусловно, более кратко, но более сложно для понимания. Мы могли бы немного упростить определение, перенеся лямбда-выражения в методы. Например, мы могли бы использовать следующие три метода вместо лямбда-выражений:

```
private String route(Object p) {
    return p.getClass().isAssignableFrom(BillingInfo.class)
```

```

    ? "BILLING_INFO"
    : "LINE_ITEMS";
}

private BillingInfo handleBillingInfo(
    BillingInfo billingInfo, MessageHeaders h) {
    // ...
}

private LineItem handleLineItems(
    LineItem lineItem, MessageHeaders h) {
    // ...
}

```

и переписать поток интеграции, используя ссылки на методы:

```

return IntegrationFlows
    ...
    .split()
    .route(
        this::route,
        mapping -> mapping
            .subFlowMapping("BILLING_INFO", sf -> sf
                .<BillingInfo> handle(this::handleBillingInfo))
            .subFlowMapping("LINE_ITEMS", sf -> sf
                .split()
                .<LineItem> handle(this::handleLineItems)));

```

В любом случае для разделения заказов используется все тот же `OrdererSplitter`. После разделения заказа его части передаются в два отдельных подпотока, согласно их типам.

10.2.6 Активаторы служб

Активаторы служб получают сообщения из входного канала и отправляют их реализации `MessageHandler`, как показано на рис. 10.7.

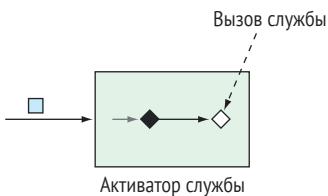


Рис. 10.7 Получив сообщение, активаторы служб вызывают некоторую службу посредством `MessageHandler`

Spring Integration предлагает несколько готовых реализаций `MessageHandler` (даже `PayloadTypeRouter` является реализацией `MessageHandler`), но на практике нередко требуется использовать свою реализацию активатора службы. В качестве примера ниже показано, как объявить компонент `MessageHandler`, настроенный как активатор службы:

```

@Bean
@ServiceActivator(inputChannel="someChannel")
public MessageHandler sysoutHandler() {
    return message -> {
        System.out.println("Message payload: " + message.getPayload());
    };
}

```

Компонент снабжен аннотацией `@ServiceActivator`, чтобы обозначить его как активатор службы, предназначенный для обработки сообщения из канала `someChannel`. Что касается самого `MessageHandler`, то он реализован в виде лямбда-выражения. Несмотря на свою простоту, эта реализация `MessageHandler` отправляет содержимое полученного сообщения в стандартный поток вывода.

Также есть возможность объявить активатор службы, который обрабатывает данные во входящем сообщении и возвращает новое содержимое, как показано в следующем фрагменте. В этом случае компонент должен реализовать интерфейс `GenericHandler`, а не `MessageHandler`.

```

@Bean
@ServiceActivator(inputChannel="orderChannel",
                   outputChannel="completeChannel")
public GenericHandler<EmailOrder> orderHandler(
    OrderRepository orderRepo) {
    return (payload, headers) -> {
        return orderRepo.save(payload);
    };
}

```

Тогда активатор службы реализует `GenericHandler` и обрабатывает сообщения с содержимым типа `EmailOrder`. Поступивший заказ сохраняется в репозитории, после чего сохраненный `EmailOrder` возвращается для отправки в выходной канал `completeChannel`.

Можно заметить, что `GenericHandler` получает не только содержимое, но и заголовки сообщений (даже притом что в примере эти заголовки никак не используются). При желании активаторы служб также можно использовать в конфигурации в стиле Java DSL, передав `MessageHandler` или `GenericHandler` в функцию `handle()`:

```

public IntegrationFlow someFlow() {
    return IntegrationFlows
        ...
        .handle(msg -> {
            System.out.println("Message payload: " + msg.getPayload());
        })
        .get();
}

```

В данном случае реализация `MessageHandler` оформлена в виде лямбда-выражения, но также можно использовать ссылку на метод или даже экземпляр класса, реализующий интерфейс `MessageHandler`.

Используя лямбда-выражение или ссылку на метод, имейте в виду, что сообщение передается в виде параметра.

Аналогично `handle()` можно написать так, чтобы он принимал `GenericHandler`, если активатор службы не предназначен для завершения потока. Применяя активатор службы сохранения заказов, представленный выше, можно настроить поток в стиле Java DSL:

```
public IntegrationFlow orderFlow(OrderRepository orderRepo) {
    return IntegrationFlows
        ...
        .<EmailOrder>handle((payload, headers) -> {
            return orderRepo.save(payload);
        })
        ...
        .get();
}
```

При использовании `GenericHandler` лямбда-выражения или ссылки на методы должны принимать параметры с содержимым и заголовками сообщения. Кроме того, если вы решите использовать `GenericHandler` в конце потока, то нужно вернуть `null`, иначе вы получите сообщение об ошибке, говорящее, что выходной канал не указан.

10.2.7 Шлюзы

Шлюз – это средство, с помощью которого приложение сможет отправить данные в поток интеграции и при необходимости получить ответ – результат работы потока. Шлюзы в Spring Integration реализуются как интерфейсы, которые приложение может вызывать для отправки сообщений в поток интеграции (рис. 10.8).

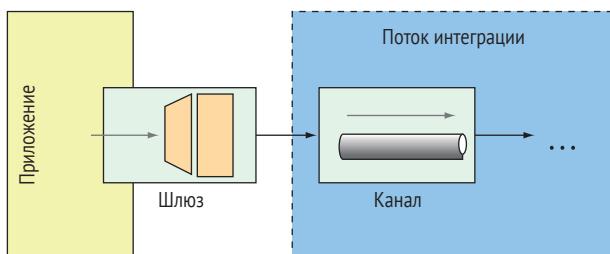


Рис. 10.8 Шлюзы – это интерфейсы, через которые приложение может отправлять сообщения в поток интеграции

Вы уже видели пример шлюза сообщений в виде `FileWriterGateway`. `FileWriterGateway` – это односторонний шлюз с методом, принимающим строку для записи в файл и возвращающим `void`. Однако без труда можно написать двунаправленный шлюз. Реализуя интерфейс шлюза, убедитесь, что метод возвращает некоторое значение для публикации в потоке интеграции.

Например, представьте шлюз, который служит входом в простой поток интеграции, принимающий строку и преобразующий ее в верхний регистр. Вот как может выглядеть интерфейс такого шлюза:

```
package sia6;
import org.springframework.integration.annotation.MessagingGateway;
import org.springframework.stereotype.Component;

@Component
@MessagingGateway(defaultRequestChannel="inChannel",
                   defaultReplyChannel="outChannel")
public interface UpperCaseGateway {
    String uppercase(String in);
}
```

Самое замечательное в этом интерфейсе – его не нужно реализовывать. Spring Integration автоматически создаст реализацию во время выполнения, которая отправляет и получает данные по указанным каналам.

Когда вызывается метод `uppercase()`, заданная строка передается в канал `inChannel` в потоке интеграции. Независимо от особенностей определения потока, когда данные поступают в канал `outChannel`, они возвращаются из метода `uppercase()`.

Преобразование в верхний регистр – это упрощенный пример потока интеграции с единственным шагом, на котором происходит преобразование строки в верхний регистр. Вот как то же самое выражается в конфигурации в стиле Java DSL:

```
@Bean
public IntegrationFlow uppercaseFlow() {
    return IntegrationFlows
        .from("inChannel")
        .<String, String> transform(s -> s.toUpperCase())
        .channel("outChannel")
        .get();
}
```

Здесь поток начинает работу с поступлением данных в канал `inChannel`. Затем содержимое сообщения обрабатывается преобразователем, который здесь определен как лямбда-выражение и производит преобразование в верхний регистр. После этого сообщение публикуется в канал `outChannel`, который мы объявили в качестве канала ответа для интерфейса `UpperCaseGateway`.

10.2.8 Адаптеры каналов

Адаптеры каналов – это точки входа и выхода в потоке интеграции. Данные входят в поток интеграции через адаптер входного канала и выходят из потока через адаптер выходного канала (рис. 10.9).



Рис. 10.9 Адаптеры каналов – это точки входа и выхода в потоке интеграции

Адаптеры входного канала могут быть разного вида, в зависимости от источника данных. Например, можно объявить адаптер входного канала, который вводит в поток увеличивающиеся числа из `AtomicInteger`¹. В конфигурации на Java это может выглядеть так:

```
@Bean
@InboundChannelAdapter(
    poller=@Poller(fixedRate="1000"), channel="numberChannel")
public MessageSource<Integer> numberSource(AtomicInteger source) {
    return () -> {
        return new GenericMessage<>(source.getAndIncrement());
    };
}
```

Этот метод объявляет bean-компонент адаптера входного канала, который, согласно аннотации `@InboundChannelAdapter`, отправляет число из внедренного `AtomicInteger` в канал `numberChannel` каждую секунду (раз в 1000 мс).

Аннотация `@InboundChannelAdapter` определяет адаптер входного канала в конфигурации на Java, а в конфигурации Java DSL для той же цели используется метод `from()`. В следующем фрагменте определения потока интеграции показано, как создать аналогичный адаптер входного канала в Java DSL:

```
@Bean
public IntegrationFlow someFlow(AtomicInteger integerSource) {
    return IntegrationFlows
        .from(integerSource, "getAndIncrement",
            c -> c.poller(Pollers.fixedRate(1000)))
        ...
        .get();
}
```

Часто адаптеры каналов предоставляются модулями конечных точек в Spring Integration. Предположим, например, что нам нужен адаптер входного канала, который наблюдает за содержимым указанного каталога и отправляет любые файлы, записанные в него, в канал `file-channel`. Для этой цели в следующей конфигурации на Java ис-

¹ `AtomicInteger` можно использовать, например, для увеличения счетчика в многопоточном окружении, когда есть вероятность одновременного поступления в канал нескольких сообщений.

пользуется `FileReadingMessageSource` из модуля конечной точки `file` в Spring Integration:

```
@Bean
@InboundChannelAdapter(channel="file-channel",
    poller=@Poller(fixedDelay="1000"))
public MessageSource<File> fileReadingMessageSource() {
    FileReadingMessageSource sourceReader = new FileReadingMessageSource();
    sourceReader.setDirectory(new File(INPUT_DIR));
    sourceReader.setFilter(new SimplePatternFileListFilter(FILE_PATTERN));
    return sourceReader;
}
```

В Java DSL аналогичный адаптер входного канала для чтения файлов можно создать вызовом метода `inboundAdapter()` класса `Files`. Как показано ниже, адаптер выходного канала – это конец потока интеграции, передающий окончательное сообщение приложению или какой-либо другой системе:

```
@Bean
public IntegrationFlow fileReaderFlow() {
    return IntegrationFlows
        .from(Files.inboundAdapter(new File(INPUT_DIR))
            .patternFilter(FILE_PATTERN))
        .get();
}
```

Активаторы служб, реализованные как обработчики сообщений, часто служат адаптерами выходных каналов, например когда данные необходимо передать самому приложению. Мы уже разбирали активаторы служб, поэтому не будем повторно обсуждать их.

Однако стоит отметить, что модули конечных точек в Spring Integration предоставляют полезные обработчики сообщений для нескольких распространенных случаев использования. Вы уже видели пример такого адаптера выходного канала, `FileWritingMessageHandler`, в листинге 10.3. Теперь, коль скоро мы коснулись конечных точек Spring Integration, давайте кратко рассмотрим, какие готовые к использованию модули конечных точек интеграции доступны.

10.2.9 Модули конечных точек

Это замечательно, что Spring Integration позволяет создавать свои адаптеры каналов. Но еще лучше, что Spring Integration предоставляет более двух десятков модулей конечных точек, содержащих готовые адаптеры каналов, – как входных, так и выходных – для интеграции с различными внешними системами, включая перечисленные в табл. 10.1.

Как следует из списка в табл. 10.1, Spring Integration предоставляет обширный набор компонентов для удовлетворения многих потребностей интеграции. Большинству приложений никогда не понадо-

бится даже часть того, что предлагает Spring Integration. Но приятно осознавать, что Spring Integration может помочь нам, если вдруг понадобится какой-либо из этих компонентов.

Таблица 10.1 Spring Integration предоставляет более двух десятков модулей конечных точек для интеграции с внешними системами

Модуль	Идентификатор артефакта зависимости (групповой идентификатор: org.springframework.integration)
AMQP	spring-integration-amqp
События приложений	spring-integration-event
Atom и RSS	spring-integration-feed
Электронная почта	spring-integration-mail
Файловая система	spring-integration-file
FTP/FTPS	spring-integration-ftp
GemFire	spring-integration-gemfire
HTTP	spring-integration-http
JDBC	spring-integration-jdbc
JMS	spring-integration-jms
JMX	spring-integration-jmx
JPA	spring-integration-jpa
Kafka	spring-integration-kafka
MongoDB	spring-integration-mongodb
MQTT	spring-integration-mqtt
R2DBC	spring-integration-r2dbc
Redis	spring-integration-redis
RMI	spring-integration-rmi
RSocket	spring-integration-rsocket
SFTP	spring-integration-sftp
STOMP	spring-integration-stomp
Stream	spring-integration-stream
Syslog	spring-integration-syslog
TCP/UDP	spring-integration-ip
WebFlux	spring-integration-webflux
Веб-службы	spring-integration-ws
WebSocket	spring-integration-websocket
XMPP	spring-integration-xmpp
ZeroMQ	spring-integration-zeromq
ZooKeeper	spring-integration-zookeeper

Однако мы не можем охватить в этой главе все адаптеры каналов, предоставляемые модулями, которые перечислены в табл. 10.1. Вы уже видели примеры использования модуля файловой системы для записи в файлы, а чуть ниже я покажу, как использовать модуль электронной почты для чтения электронных писем.

Все модули конечных точек предлагают адаптеры каналов, которые можно объявить как компоненты в конфигурации на Java или слаться на них через статические методы в конфигурации Java DSL.

Я советую изучить все другие модули конечных точек, которые вас заинтересуют. Вы обнаружите, что все они используются более или менее одинаково. А теперь обратим наше внимание на модуль конечной точки электронной почты и посмотрим, как можно использовать его в приложении Taco Cloud.

10.3 Создание потока интеграции для электронной почты

Итак, представим, что в Taco Cloud решили дать своим клиентам возможность отправлять свои рецепты тако и размещать заказы по электронной почте. Они провели рекламную кампанию, разослав листовки и разместив объявления в газетах, приглашая всех присыпать заказы на тако по электронной почте. И эта кампания увенчалась грандиозным успехом! К сожалению, слишком грандиозным. На электронную почту стало приходить так много писем, что пришлось нанять дополнительных сотрудников, которые только читают электронные письма и пересыпают заказы в систему заказов.

В этом разделе мы реализуем поток интеграции, который проверяет почтовый ящик Taco Cloud на наличие электронных писем с заказами, анализирует электронные письма, извлекает из них сведения о заказе и пересыпает заказы в систему Taco Cloud для обработки. Проще говоря, поток интеграции будет использовать адаптер входного канала из модуля конечной точки электронной почты для приема электронных писем в поток интеграции.

Следующим шагом в потоке интеграции будет анализ электронных писем и создание объектов заказов для передачи другому обработчику, отправляющему заказы в Taco Cloud REST API, где они будут обработаны как обычно. Для начала определим простой класс конфигурационных свойств, чтобы зафиксировать особенности обработки электронных писем:

```
package tacos.email;

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

@Data
@ConfigurationProperties(prefix="tacocloud.email")
@Component
public class EmailProperties {

    private String username;
    private String password;
    private String host;
    private String mailbox;
```

```

private long pollRate = 30000;

public String getImapUrl() {
    return String.format("imaps://\%s:\%s@\%s/\%s",
        this.username, this.password, this.host, this.mailbox);
}

}

```

Как видите, `EmailProperties` хранит свойства, которые используются для создания IMAP URL. Поток использует этот URL для подключения к почтовому серверу Taco Cloud и извлечения электронных писем. Среди хранимых свойств – имя пользователя электронной почты и пароль, а также имя хоста сервера IMAP, почтовый ящик и частота опроса (по умолчанию опрос производится раз в 30 секунд).

Класс `EmailProperties` снабжен аннотацией `@ConfigurationProperties` с атрибутом `prefix`, которому присвоено значение `tacocloud.email`. Это означает, что мы можем настроить детали использования электронной почты в файле `application.yml`:

```

tacocloud:
  email:
    host: imap.tacocloud.com
    mailbox: INBOX
    username: taco-in-flow
    password: 1l0v3T4c0s
    poll-rate: 10000

```

Конечно, эта конфигурация сервера электронной почты – вымышленная. Вам нужно будет изменить ее у себя, чтобы она соответствовала фактическим параметрам вашего почтового сервера.

Кроме того, вы можете получить предупреждение о «неизвестном свойстве» в вашей IDE. Это связано с тем, что IDE ищет метаданные, чтобы понять, что означают эти свойства. Предупреждения не нарушают работу фактического кода, и вы можете игнорировать их, если хотите. Или можете убрать их, добавив в спецификацию сборки следующую зависимость (также доступную в Spring Initializr в виде флагка **Spring Configuration Processor** (Процессор конфигурации Spring)):

```

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

```

Эта зависимость включает поддержку автоматического создания метаданных для конфигурационных свойств, подобных тем, что мы используем для настройки сведений о сервере электронной почты.

Теперь воспользуемся классом `EmailProperties` для настройки потока интеграции. Наш поток будет немного похож на изображенный на рис. 10.10.

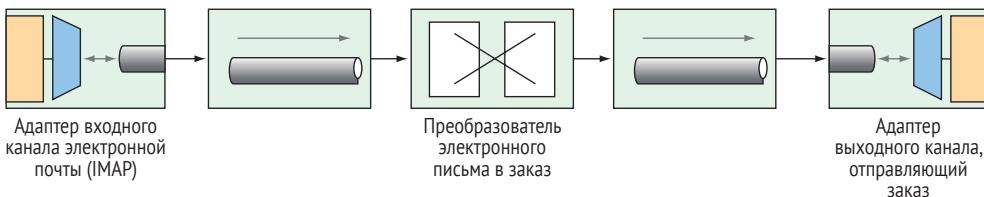


Рис. 10.10 Поток интеграции для приема заказов по электронной почте

Определить этот поток можно двумя способами:

- *в самом приложении Taco Cloud*; в конце потока активатор службы отправит запрос указанному репозиторию для создания заказа;
- *как отдельное приложение*; в конце потока активатор службы отправит запрос POST в Taco Cloud API, чтобы передать заказ.

Выбор того или иного варианта не имеет большого значения для самого потока, кроме способа реализации активатора службы. Но поскольку нам понадобятся некоторые типы, представляющие рецепты, заказы и ингредиенты, которые немного отличаются от тех, что мы уже определили в основном приложении Taco Cloud, мы реализуем поток интеграции в отдельном приложении, чтобы избежать путаницы с существующими прикладными типами.

Мы также можем выбирать между определением потока в виде конфигурации XML, на Java или Java DSL. Мне больше нравится элегантность Java DSL, поэтому использую этот подход. Вы же можете использовать любой другой способ, если вам нравятся дополнительные сложности. А пока давайте взглянем на конфигурацию Java DSL, представленную в листинге 10.5.

Листинг 10.5 Определение потока интеграции для приема электронных писем и пересылки заказов, извлеченных из них

```
package tacos.email;

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.mail.dsl.Mail;

@Configuration
public class TacoOrderEmailIntegrationConfig {

    @Bean
    public IntegrationFlow tacoOrderEmailFlow(
        EmailProperties emailProps,
        EmailToOrderTransformer emailToOrderTransformer,
        OrderSubmitMessageHandler orderSubmitHandler) {
```

```

        return IntegrationFlows
            .from(Mail imapInboundAdapter(emailProps.getImapUrl()),
                  e -> e.poller(
                      Pollers.fixedDelay(emailProps.getPollRate())))
            .transform(emailToOrderTransformer)
            .handle(orderSubmitHandler)
            .get();
    }

}

```

Поток интеграции с электронной почтой, определяемый в методе `tacoOrderEmailFlow()`, включает три отдельных компонента:

- *адаптер входного канала электронной почты IMAP* – этот адаптер создается с IMAP URL, сгенерированным методом `getImapUrl()` объекта `EmailProperties`, и проверяет наличие электронных писем через регулярные интервалы времени, как определено в свойстве `pollRate` объекта `EmailProperties`. Входящие электронные письма передаются в канал, соединяющий его с преобразователем;
- *преобразователь, превращающий электронное письмо в объект заказа* – преобразователь реализован в классе `EmailToOrderTransformer`, который внедряется в метод `tacoOrderEmailFlow()`. Заказы, созданные преобразователем, передаются конечному компоненту через еще один канал;
- *обработчик (действующий как адаптер выходного канала)* – принимает объект заказа и отправляет его в Taco Cloud REST API.

Мы имеем возможность вызова `Mail imapInboundAdapter()` благодаря включению модуля конечной точки `Email` в спецификацию сборки проекта. Вот как выглядит зависимость Maven:

```

<dependency>
    <groupId>org.springframework.integration</groupId>
    <artifactId>spring-integration-mail</artifactId>
</dependency>

```

Класс `EmailToOrderTransformer` – это реализация интерфейса `Transformer` из Spring Integration путем расширения `AbstractMailMessageTransformer` (как показано в листинге 10.6).

Листинг 10.6 Преобразование входящих электронных писем в заказы с использованием преобразователя

```

package tacos.email;

import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import javax.mail.Message;
import javax.mail.MessagingException;

```

```
import javax.mail.internet.InternetAddress;
import org.apache.commons.text.similarity.LevenshteinDistance;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.integration.mail.transformer
    .AbstractMailMessageTransformer;
import org.springframework.integration.support
    .AbstractIntegrationMessageBuilder;
import org.springframework.integration.support.MessageBuilder;
import org.springframework.stereotype.Component;

@Component
public class EmailToOrderTransformer
    extends AbstractMailMessageTransformer<EmailOrder> {

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

    private static final String SUBJECT_KEYWORDS = "TACO ORDER";

    @Override
    protected AbstractIntegrationMessageBuilder<EmailOrder>
        doTransform(Message mailMessage) throws Exception {
        EmailOrder tacoOrder = processPayload(mailMessage);
        return MessageBuilder.withPayload(tacoOrder);
    }

    private EmailOrder processPayload(Message mailMessage) {
        try {
            String subject = mailMessage.getSubject();
            if (subject.toUpperCase().contains(SUBJECT_KEYWORDS)) {
                String email =
                    ((InternetAddress) mailMessage.getFrom()[0]).getAddress();
                String content = mailMessage.getContent().toString();
                return parseEmailToOrder(email, content);
            }
        } catch (MessagingException e) {
            log.error("MessagingException: {}", e);
        } catch (IOException e) {
            log.error("IOException: {}", e);
        }
        return null;
    }

    private EmailOrder parseEmailToOrder(String email, String content) {
        EmailOrder order = new EmailOrder(email);
        String[] lines = content.split("\r?\n");
        for (String line : lines) {
            if (line.trim().length() > 0 && line.contains(":")) {
                String[] lineSplit = line.split(":");
                String tacoName = lineSplit[0].trim();
                String ingredients = lineSplit[1].trim();
                String[] ingredientsSplit = ingredients.split(",");
                order.setTacoName(tacoName);
                order.setIngredients(ingredientsSplit);
            }
        }
        return order;
    }
}
```

```

List<String> ingredientCodes = new ArrayList<>();
for (String ingredientName : ingredientsSplit) {
    String code = lookupIngredientCode(ingredientName.trim());
    if (code != null) {
        ingredientCodes.add(code);
    }
}
Taco taco = new Taco(tacoName);
taco.setIngredients(ingredientCodes);
order.addTaco(taco);
}
return order;
}

private String lookupIngredientCode(String ingredientName) {
    for (Ingredient ingredient : ALL_INGREDIENTS) {
        String ucIngredientName = ingredientName.toUpperCase();
        if (LevenshteinDistance.getDefaultInstance()
            .apply(ucIngredientName, ingredient.getName()) < 3 ||
            ucIngredientName.contains(ingredient.getName()) ||
            ingredient.getName().contains(ucIngredientName)) {
            return ingredient.getCode();
        }
    }
    return null;
}

private static Ingredient[] ALL_INGREDIENTS = new Ingredient[] {
    new Ingredient("FLTO", "FLOUR TORTILLA"),
    new Ingredient("COTO", "CORN TORTILLA"),
    new Ingredient("GRBF", "GROUND BEEF"),
    new Ingredient("CARN", "CARNITAS"),
    new Ingredient("TMTO", "TOMATOES"),
    new Ingredient("LETC", "LETTUCE"),
    new Ingredient("CHED", "CHEDDAR"),
    new Ingredient("JACK", "MONTERREY JACK"),
    new Ingredient("SLSA", "SALSA"),
    new Ingredient("SRCR", "SOUR CREAM")
};
}

```

`AbstractMailMessageTransformer` – это базовый класс для обработки сообщений, содержащих электронные письма. Он извлекает из входящего сообщения информацию об электронном письме в объект `Message`, который затем передается в метод `doTransform()`.

В методе `doTransform()` мы передаем `Message` приватному методу `processPayload()`, чтобы преобразовать электронное письмо в объект `EmailOrder`. Несмотря на некоторое сходство, объект `EmailOrder` все же отличается от объекта `TacoOrder`, используемого в основном приложении `Taco Cloud`, – он несколько проще, как показано ниже:

```

package tacos.email;

import java.util.ArrayList;
import java.util.List;
import lombok.Data;

@Data
public class EmailOrder {

    private final String email;
    private List<Taco> tacos = new ArrayList<>();

    public void addTaco(Taco taco) {
        this.tacos.add(taco);
    }

}

```

Экземпляр `EmailOrder` несет не всю информацию, необходимую для доставки и оплаты, а только электронное письмо клиента, полученное из почтового ящика.

Превращение писем в заказы – сложная задача. Даже простейшая реализация включает несколько десятков строк кода. И эти несколько десятков строк кода никак не способствуют обсуждению Spring Integration и реализации преобразователя. Поэтому для экономии места я опущу детали метода `processPayload()`.

В заключение `EmailToOrderTransformer` возвращает `MessageBuilder` с объектом `EmailOrder`. Сообщение, созданное `MessageBuilder`, отправляется последнему компоненту в потоке интеграции: обработчику сообщений, который пересыпает заказ в Taco Cloud API. `OrderSubmitMessageHandler`, как показано в листинге 10.7, реализует интерфейс `GenericHandler` и обрабатывает сообщения, содержащие экземпляры `EmailOrder`.

Листинг 10.7 Отправка заказов в Taco Cloud API через обработчик сообщений

```

package tacos.email;

import org.springframework.integration.handler.GenericHandler;
import org.springframework.messaging.MessageHeaders;
import org.springframework.stereotype.Component;
import org.springframework.web.client.RestTemplate;

@Component
public class OrderSubmitMessageHandler
    implements GenericHandler<EmailOrder> {

    private RestTemplate rest;
    private ApiProperties apiProps;

    public OrderSubmitMessageHandler(ApiProperties apiProps, RestTemplate rest) {

```

```

    this.apiProps = apiProps;
    this.rest = rest;
}

@Override
public Object handle(EmailOrder order, MessageHeaders headers) {
    rest.postForObject(apiProps.getUrl(), order, String.class);
    return null;
}
}

```

Чтобы удовлетворить требования интерфейса `GenericHandler`, класс `OrderSubmitMessageHandler` переопределяет метод `handle()`. Этот метод получает объект `EmailOrder` и использует внедренный компонент `RestTemplate` для отправки `EmailOrder` в запросе POST на URL, хранящийся во внедренном объекте `ApiProperties`. Кроме того, метод `handle()` возвращает `null`, чтобы сообщить, что этот обработчик завершает поток.

Мы используем `ApiProperties`, дабы избежать необходимости жестко определять URL в вызове `postForObject()`. Вот как определяются эти конфигурационные свойства:

```

package tacos.email;

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

@Data
@ConfigurationProperties(prefix = "tacocloud.api")
@Component
public class ApiProperties {
    private String url;
}

```

Вот как можно настроить URL для Taco Cloud API в `application.yml`:

```
tacocloud:
  api:
    url: http://localhost:8080/orders/fromEmail
```

Чтобы сделать `RestTemplate` доступным для внедрения в `OrderSubmitMessageHandler`, нужно добавить в спецификацию сборки следующую начальную зависимость:

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

Это не только обеспечит доступность `RestTemplate` в пути к классам, но также заставит механизм автоконфигурации выполнить на-

стройку Spring MVC. Автономному приложению, играющему роль потока интеграции Spring Integration, не требуется Spring MVC или встроенный сервер Tomcat, который настроит механизм автоконфигурации. Поэтому мы должны отключить автоконфигурацию Spring MVC в *application.yml*:

```
spring:  
  main:  
    web-application-type: none
```

Свойству `spring.main.web-application-type` можно также присвоить значение `servlet`, `reactive` или `none`. Когда Spring MVC находится в пути к классам, автоконфигурация устанавливает значение `servlet`. Но здесь мы задали значение `none`, чтобы Spring MVC и Tomcat не настраивались автоматически. (О реактивных веб-приложениях мы поговорим в главе 12.)

Итоги

- Spring Integration позволяет определять потоки для обработки данных, входящих в приложение или исходящих из него.
- Потоки интеграции можно определять в формате XML, на Java или с использованием более компактного стиля конфигурации Java DSL.
- Шлюзы сообщений и адаптеры каналов действуют как точки входа и выхода потока интеграции.
- Сообщения могут преобразовываться, ветвиться, агрегироваться, маршрутизироваться и обрабатываться активаторами служб.
- Каналы сообщений соединяют компоненты потока интеграции.

Часть III

Реактивный Spring

В третьей части книги мы исследуем поддержку реактивного программирования в Spring. В главе 11 обсудим основы реактивного программирования с использованием Project Reactor – библиотеки реактивного программирования, лежащей в основе реактивных возможностей Spring. Затем исследуем некоторые из наиболее полезных реактивных операций Reactor. В главе 12 мы вернемся к разработке REST API и познакомимся с веб-фреймворком Spring WebFlux, многое позаимствовавшим из Spring MVC и предлагающим новую реактивную модель программирования для разработки веб-приложений. В главе 13 мы посмотрим, как организовать хранение реактивных данных в базах данных Cassandra и Mongo с использованием Spring Data. В главе 14, завершающей третью часть, мы рассмотрим RSocket – коммуникационный протокол, реализующий реактивную альтернативу HTTP.

1 1

Введение в Reactor

В этой главе рассматриваются следующие темы:

- знакомство с реактивным программированием;
- Project Reactor;
- реактивные операции с данными.

Приходилось ли вам когда-нибудь подписываться на газеты или журналы? Интернет отнял часть подписчиков у традиционных изданий, но было время, когда подписка на газеты была одним из лучших способов оставаться в курсе последних событий. Вы могли рассчитывать на свежие новости о текущих событиях каждое утро и читать их во время завтрака или по дороге на работу.

Теперь предположим, что после оплаты подписки прошло несколько дней, а газеты не доставляются. Проходит еще несколько дней, и вы звоните в отделение связи и спрашиваете, почему до сих пор не было доставлено ни одного номера ежедневной газеты. Каково же будет ваше удивление, если вам ответят: «Вы заплатили за целый год. Год еще не закончился. Вы обязательно получите их все, как только будет готов полный годовой выпуск газет».

К счастью, подписка на прессу работает совсем не так. Газеты имеют определенный срок актуальности. Они доставляются с максимальной скоростью после выпуска, чтобы их можно было прочитать, пока их содержание еще свежее. Кроме того, пока вы читаете последний

номер, газетные репортеры пишут новые истории для будущих выпусков, а издательство готовит к печати следующий выпуск – и все это параллельно.

При разработке приложений мы можем писать код, используя два стиля – императивный и реактивный, которые можно охарактеризовать так:

- *императивный* код очень похож на абсурдную гипотетическую подпиську на газету. Это набор задач, выполняющихся последовательно, каждая из которых выполняется строго после предыдущей задачи. Данные обрабатываются большими порциями и не могут передаваться следующей задаче, пока предыдущая задача не обработает их до конца;
- *реактивный* код очень похож на реальную подписку на газету. Имеется определенный круг задач для обработки данных, но эти задачи могут выполняться параллельно. Каждая может обрабатывать подмножества данных, передавать их следующей задаче в очереди и сразу приступать к обработке следующего подмножества данных.

В этой главе мы временно отойдем от приложения Taco Cloud и познакомимся с Project Reactor (<https://projectreactor.io/>) – библиотекой реактивного программирования, входящей в семейство проектов Spring. И поскольку она служит основой поддержки реактивного программирования в Spring, очень важно, чтобы вы поняли, как работать с Reactor, прежде чем перейти к изучению особенностей создания реактивных контроллеров и репозиториев с помощью Spring. Но прежде чем начать использовать Reactor, мы кратко познакомимся с основами реактивного программирования.

11.1 Основы реактивного программирования

Реактивное программирование – это парадигма, альтернатива императивному программированию. Эта альтернатива появилась, потому что реактивное программирование устраняет ограничения, свойственные императивному программированию. Поняв эти ограничения, вы будете лучше понимать преимущества реактивной модели.

ПРИМЕЧАНИЕ Реактивное программирование не панацея. Ни из этой главы, ни из какого другого обсуждения реактивного программирования не следует делать вывод, что императивное программирование – это зло, а реактивное программирование – ваше все. Как и все, что используется в разработке программного обеспечения, реактивное программирование идеально подходит в одних случаях и никуда не годится в других. Не забывайте о pragmatizme.

Подавляющее большинство разработчиков, в том числе и я, быстро осваивают императивное программирование и большую часть (или весь) своего кода пишут в императивном стиле. Императивное программирование достаточно понятное и логичное. Школьники с легкостью осваивают его на уроках информатики. И оно достаточно мощное, чтобы позволить сформировать основную часть кода, используемого на крупных предприятиях.

Идея проста: вы пишете код в виде списка инструкций, которые компьютер должен выполнить в том порядке, в каком они записаны. Пока задача выполняется, программа ждет ее завершения и только потом передает управление следующей задаче. На каждом этапе обрабатываемые данные должны быть доступны полностью, чтобы их можно было обработать как единое целое.

Это хорошо... до определенного момента. На время выполнения задачи – особенно если это задача ввода/вывода, такая как запись информации в базу данных или выборка данных с удаленного сервера, – поток выполнения блокируется и не может делать ничего другого. Проще говоря, заблокированный поток выполнения впустую расходует ресурсы компьютера.

Большинство языков программирования, включая Java, поддерживают параллельное программирование. В Java легко запустить другой поток выполнения и направить его для выполнения какой-то иной работы, пока вызывающий поток занимается чем-то другим. Но эта простота мало что дает, потому что потоки, скорее всего, сами заблокируются. Управление параллельным выполнением нескольких потоков выполнения – сложная задача. И чем больше потоков, тем сложнее.

Реактивное программирование, напротив, является функциональным и декларативным по своей природе. Вместо описания набора шагов, которые должны выполняться последовательно, реактивное программирование предполагает описание конвейера или потока данных (stream). Реактивный поток данных не требует, чтобы данные были доступны и обрабатывались как единое целое, а обрабатывает их по мере появления. На самом деле входные данные могут быть бесконечными (например, постоянный поток данных о температуре, измеряемой в масштабе реального времени).

ПРИМЕЧАНИЕ Если вы незнакомы с функциональным программированием на Java, то вам может быть интересно познакомиться с книгами Пьера-Ива Сомона (Pierre-Yves Saumont) «Functional Programming in Java» (Manning, 2017) или Михала Плахты (Michał Płachta) «Grokking Functional Programming» («Функциональное программирование на языке Grokking» (Manning, 2021)).

Проводя аналогию с реальным миром, императивное программирование можно представить в виде воздушного шарика, наполненно-

го водой, а реактивное программирование – в виде шланга. Оба можно использовать, чтобы подшутить над другом в жаркий летний день. Но по стилю исполнения они различаются:

- шарик с водой выплескивает всю воду на намеченную цель в момент удара. Однако вместимость шарика ограничена, и если вы решите подшутить над несколькими друзьями, то ваш единственный выбор – наполнить водой несколько шариков;
- садовый шланг несет воду в виде потока, который течет от одного конца шланга к другому. Емкость садового шланга конечна в любой конкретный момент времени, но она не ограничена в ходе водной битвы. Пока вода поступает в шланг, она будет продолжать течь по шлангу и распыляться из сопла. Одним и тем же шлангом легко намочить столько друзей, сколько вы пожелаете.

В шариках с водой (императивном программировании) нет ничего плохого по своей сути, но человек, держащий садовый шланг (применяющий реактивное программирование), имеет преимущество с точки зрения масштабируемости и производительности.

11.1.1 Определение реактивных потоков данных

Reactive Streams – это инициатива, запущенная в конце 2013 года инженерами из Netflix, Lightbend и Pivotal (последняя компания в этом списке как раз и занимается разработкой Spring). Цель Reactive Streams – поддержка стандарта асинхронной обработки потоков с не-блокирующим обратным давлением (backpressure).

Мы уже коснулись асинхронной природы реактивного программирования; она позволяет выполнять задачи параллельно и достигать большей масштабируемости. Обратное давление – это средство, помогающее получателям данных избежать перегрузки при работе со слишком быстрым источником данных; с его помощью получатели могут установить ограничение на объем, который они готовы обработать в единицу времени.

Потоки Java и Reactive Streams

Между потоками Java и Reactive Streams много общего. Начнем с того, что в их именах есть слово «потоки». Также оба предоставляют функциональный API для работы с данными. На самом деле, как вы увидите ниже, они даже используют множество похожих операций.

Однако потоки Java обычно синхронны и работают с конечным набором данных. По сути, это средство перебора содержимого коллекции с функциями.

Reactive Streams поддерживает асинхронную обработку наборов данных любого размера, включая бесконечные наборы данных. Обработка данных выполняется в режиме реального времени по мере их поступления с учетом обратного давления, помогающего не перегружать получателей.

С другой стороны, Flow API в JDK 9 соответствуют Reactive Streams. Типы `Flow.Publisher`, `Flow.Subscriber`, `Flow.Subscription` и `Flow.Processor` в JDK 9 прямо соответствуют типам `Publisher`, `Subscriber`, `Subscription` и `Processor` в Reactive Streams. Тем не менее Flow API в JDK 9 не является реальной реализацией Reactive Streams.

Спецификацию Reactive Streams можно свести к четырем определениям интерфейсов: `Publisher`, `Subscriber`, `Subscription` и `Processor`. Издатель `Publisher` создает данные и отправляет их подписчику `Subscriber`. Интерфейс издателя `Publisher` объявляет единственный метод `subscribe()`, с помощью которого подписчик `Subscriber` может подписаться на события издателя:

```
public interface Publisher<T> {
    void subscribe(Subscriber<? super T> subscriber);
}
```

После оформления подписки подписчик `Subscriber` может получать события от издателя. Эти события отправляются через методы в интерфейсе подписчика `Subscriber`:

```
public interface Subscriber<T> {
    void onSubscribe(Subscription sub);
    void onNext(T item);
    void onError(Throwable ex);
    void onComplete();
}
```

Первое событие, которое получит подписчик, – это вызов `onSubscribe()`. Когда издатель вызывает `onSubscribe()`, он передает объект `Subscription` подписчику. Именно через `Subscription` подписчик `Subscriber` может управлять своей подпиской:

```
public interface Subscription {
    void request(long n);
    void cancel();
}
```

Подписчик `Subscriber` может вызвать `request()`, чтобы запросить отправку данных, или `cancel()`, чтобы отменить подписку. При вызове `request()` подписчик передает значение типа `long`, обозначающее количество элементов данных, которое он готов принять. Именно здесь возникает обратное давление, не позволяющее издателю отправлять больше данных, чем может обработать подписчик. После получения запрошенного количества элементов подписчик может снова вызвать `request()` и запросить дополнительные данные.

После запроса данных подписчиком `Subscriber` они начинают передаваться через поток. Для каждого элемента, опубликованного издателем `Publisher`, будет вызываться метод `onNext()` для доставки дан-

ных подписчику `Subscriber`. В случае появления какой-либо ошибки вызывается `onError()`. Если у издателя `Publisher` закончились данные и он не собирается производить новые данные, он вызывает метод `onComplete()`, чтобы сообщить подписчику о завершении работы.

Теперь перейдем к интерфейсу `Processor`. Он представляет комбинацию подписчика `Subscriber` и издателя `Publisher`:

```
public interface Processor<T, R>
    extends Subscriber<T>, Publisher<R> {}
```

Так же как подписчик `Subscriber`, процессор `Processor` будет получать и каким-то образом обрабатывать данные. Затем он сменит ориентацию и будет действовать как издатель `Publisher`, чтобы опубликовать новые данные для своих подписчиков.

Как видите, спецификация Reactive Streams довольно проста. Она позволяет легко создавать конвейеры обработки данных, в которых издатели перекачивают данные через ноль или более процессоров и далее конечному подписчику.

Однако интерфейсы Reactive Streams не позволяют составлять такие конвейеры функциональным способом. По этой причине был создан проект Project Reactor – реализация спецификации Reactive Streams, предоставляющая функциональный API для создания реактивных потоков. Как будет показано в следующих главах, Reactor образует фундамент модели реактивного программирования в Spring. В оставшейся части этой главы мы исследуем Project Reactor и, надеюсь, весело проведем время.

11.2 Reactor

Реактивное программирование требует иного мышления, отличного от принятого в императивном программировании. Вместо описания последовательности шагов, которые необходимо выполнить, реактивное программирование предполагает определение конвейера, по которому будут проходить данные. По мере прохождения по конвейеру данные могут изменяться или использоваться каким-то образом.

Например, предположим, что мы решили преобразовать в верхний регистр все буквы в имени человека, вставить его в приветственное сообщение и, наконец, вывести его. В императивной модели программирования код будет выглядеть примерно так:

```
String name = "Craig";
String capitalName = name.toUpperCase();
String greeting = "Hello, " + capitalName + "!";
System.out.println(greeting);
```

В императивной модели строки кода выполняются последовательно, друг за другом, и обязательно в одном и том же потоке выпол-

нения. Никакой следующий шаг не выполнится, пока не закончится предыдущий.

Функциональный реактивный код, напротив, может решить ту же задачу, как показано здесь:

```
Mono.just("Craig")
    .map(n -> n.toUpperCase())
    .map(cn -> "Hello, " + cn + "!")
    .subscribe(System.out::println);
```

Не беспокойтесь пока о деталях этого примера; мы вскоре подробно рассмотрим операции `just()`, `map()` и `subscribe()`. В данный момент важно понять, что, несмотря на сходство с пошаговой моделью, в действительности этот реактивный пример представляет конвейер, по которому проходят данные. На каждом этапе конвейера данные каким-то образом обрабатываются, но нельзя сделать никаких предположений о том, в каком потоке выполнения будут производиться те или иные операции. Они могут быть выполнены в одном потоке... или в разных.

Моно в этом примере – это один из двух основных типов Reactor. Второй базовый тип – Flux. Оба являются реактивными реализациями Publisher. Тип Flux представляет конвейер из произвольного количества (от нуля до бесконечности) элементов данных. Mono – это специализированный реактивный тип, оптимизированный для случаев, когда известно, что набор данных содержит не более одного элемента данных.

Reactor и RxJava (ReactiveX)

Знакомым с RxJava или ReactiveX может показаться, что Mono и Flux очень похожи на `Observable` и `Single`. На самом деле они почти эквивалентны семантически. Они даже предлагают множество одинаковых операций.

В этой книге мы будем рассматривать только Reactor, но, возможно, вам будет интересно узнать, что типы Reactor и RxJava можно преобразовывать между собой. Более того, как будет показано в следующих главах, Spring может работать и с типами RxJava.

Предыдущий пример фактически содержит три объекта Mono. Первый из них создает операция `just()`. Когда Mono выдает значение, оно передается операции `map()`, которая преобразует символы в верхний регистр и создает второй объект Mono. Когда второй экземпляр Mono публикует свои данные, выполняется вторая операция `map()`, объединяющая строки в одно целое и создающая третий объект Mono. Наконец, вызов `subscribe()` оформляет подписку на получение данных от этого объекта Mono, получает данные и выводит их.

11.2.1 Диаграммы реактивных потоков

Реактивные потоки часто иллюстрируются мраморными диаграммами (marble diagram). В своей простейшей форме мраморные диаграммы представляют временную шкалу данных, протекающих сначала через Flux или Mono (вверху), затем через некоторую операцию (в середине) и через конечный Flux или Mono (внизу). На рис. 11.1 показано, как выглядит мраморная диаграмма для Flux. Как видите, когда данные проходят через начальный объект Flux, они обрабатываются с помощью некоторой операции, и в результате создается новый экземпляр Flux, через который проходят обработанные данные.

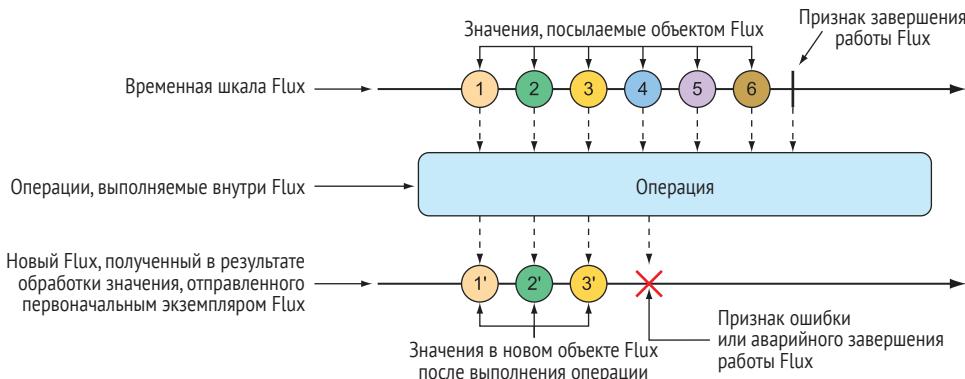


Рис. 11.1 Мраморная диаграмма, иллюстрирующая простой поток данных через Flux

На рис. 11.2 показана аналогичная мраморная диаграмма для Mono. Как видите, основное отличие состоит в том, что Mono выдаст не более одного элемента данных или признак ошибки.

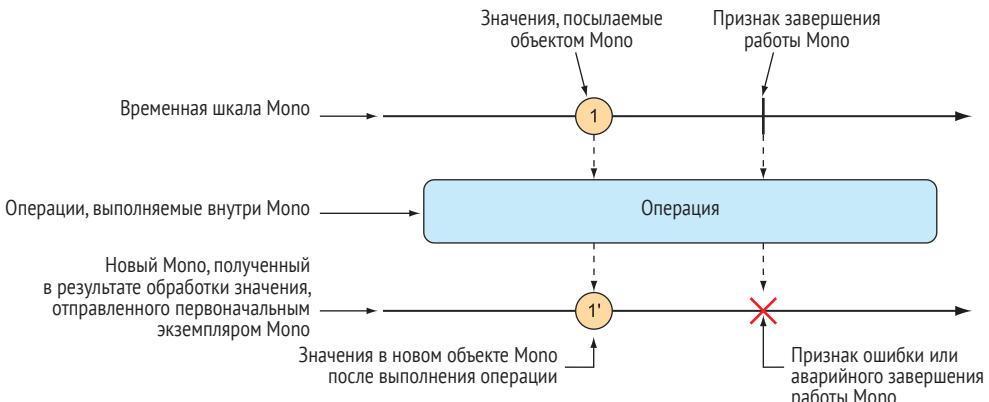


Рис. 11.2 Мраморная диаграмма, иллюстрирующая простой поток данных через Mono

В разделе 11.3 мы рассмотрим основные операции, поддерживающие Flux и Mono, и будем использовать мраморные диаграммы для визуализации их работы.

11.2.2 Добавление зависимости от Reactor

Чтобы добавить поддержку Reactor в приложение, нужно включить следующую зависимость в спецификацию сборки:

```
<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-core</artifactId>
</dependency>
```

Reactor также имеет отличную поддержку тестирования. Если вы собираетесь писать много тестов для своего кода Reactor, то обязательно добавьте следующую зависимость:

```
<dependency>
  <groupId>io.projectreactor</groupId>
  <artifactId>reactor-test</artifactId>
  <scope>test</scope>
</dependency>
```

При добавлении этих зависимостей в проект Spring Boot с поддержкой автоматическим управлением зависимостями элемент `<version>` можно опустить. Но при использовании Reactor в проекте, отличном от Spring Boot, необходимо настроить спецификацию Reactor в сборке. Следующие строки добавят в сборку выпуск Reactor 2020.0.4:

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>io.projectreactor</groupId>
      <artifactId>reactor-bom</artifactId>
      <version>2020.0.4</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Примеры, представленные в этой главе, не связаны с нашими проектами Taco Cloud. Поэтому для экспериментов лучше создать новый проект Spring с зависимостями Reactor в спецификации сборки.

Теперь, после включения Reactor в спецификацию сборки проекта, можно приступать к созданию реактивных конвейеров на основе Mono и Flux. В оставшейся части этой главы мы рассмотрим несколько операций, предлагаемых Mono и Flux.

11.3 Использование распространенных реактивных операций

Flux и Mono – наиболее важные строительные блоки в составе Reactor, а операции, предлагаемые этими двумя реактивными типами, являются раствором, связывающим их вместе в единый конвейер, по которому могут проходить данные. Flux и Mono предлагают более 500 операций, которые можно условно разделить на следующие категории:

- создание;
- комбинирование;
- преобразование;
- логика.

Как бы ни было любопытно опробовать каждую из 500 операций, чтобы посмотреть, как они работают, для этого в данной главе просто не хватит места. В этом разделе я покажу лишь несколько наиболее часто используемых операций. Начнем с операций создания.

ПРИМЕЧАНИЕ Хочу сделать небольшое замечание о примерах, демонстрирующих использование Mono. Потоки Mono и Flux поддерживают множество одинаковых операций, поэтому в большинстве случаев нет необходимости показывать одну и ту же операцию дважды, один раз для Mono и один раз для Flux. Более того, несмотря на всю полезность, рассматривать операции Mono не так интересно. Большинство примеров, демонстрируемых далее, основаны на использовании Flux. Просто знайте, что в Mono нередко можно найти эквивалентные операции.

11.3.1 Создание реактивных типов

Часто для работы с реактивными типами в Spring используется экземпляр Flux или Mono из репозитория или службы, поэтому их не нужно создавать самостоятельно. Но иногда действительно бывает необходимо создать свой экземпляр реактивного издателя.

Reactor предоставляет несколько операций для создания Flux или Mono. В этом разделе мы рассмотрим несколько наиболее часто используемых из их числа.

Создание из объектов

Если в программе есть один или несколько объектов, из которых хотелось бы создать экземпляр Flux или Mono, то можно использовать статический метод `just()`, возвращающий реактивный тип, данные в котором будут представлены прикладными объектами. Например, следующий тестовый метод создает Flux из пяти объектов типа `String`:

```
@Test
public void createAFlux_just() {
    Flux<String> fruitFlux = Flux
        .just("Apple", "Orange", "Grape", "Banana", "Strawberry");
}
```

Этот метод создаст экземпляр Flux, но у него нет подписчиков. Без подписчиков данные никуда не будут передаваться. Если вспомнить аналогию с садовым шлангом, то этот метод подсоединяет садовый шланг к крану, по которому подается вода, но пока вы не откроете кран, вода не потечет. Подписка на реактивный тип есть процедура включения потока данных.

Чтобы добавить подписчика, можно вызвать метод `subscribe()`:

```
fruitFlux.subscribe(
    f -> System.out.println("Here's some fruit: " + f)
);
```

Лямбда-выражение, что передается в вызов `subscribe()`, на самом деле является экземпляром `java.util.Consumer`, на основе которого Reactive Streams создаст подписчика `Subscriber`. Сразу после вызова `subscribe()` по конвейеру начинают поступать данные. В этом примере нет промежуточных операций, поэтому данные идут напрямую от Flux к подписчику `Subscriber`.

Вывод в консоль элементов, посылаемых экземпляром Flux или Mono, – хороший способ увидеть, как действует реактивный тип. Но еще лучше для тестирования Flux или Mono использовать `StepVerifier`. `StepVerifier` подписывается на реактивный тип Flux или Mono, а затем применяет проверки к данным, когда они проходят через поток, и в заключение убеждается, что поток завершился должным образом.

Например, чтобы убедиться, что данные проходят через `fruitFlux`, можете написать такой тест:

```
StepVerifier.create(fruitFlux)
    .expectNext("Apple")
    .expectNext("Orange")
    .expectNext("Grape")
    .expectNext("Banana")
    .expectNext("Strawberry")
    .verifyComplete();
```

В этом случае `StepVerifier` подписывается на Flux, а затем проверяет каждый элемент на соответствие ожидаемому имени фрукта. Наконец, он проверяет, что после отправки `Strawberry` объект Flux завершает работу как должно.

В остальных примерах в этой главе мы будем использовать `StepVerifier` для создания тестов, которые проверят поведение реактивного типа и помогут нам понять, как он работает.

Создание из коллекций

Объект Flux также можно создать из массива, экземпляра Iterable или Stream. На рис. 11.3 показана соответствующая мраморная диаграмма.

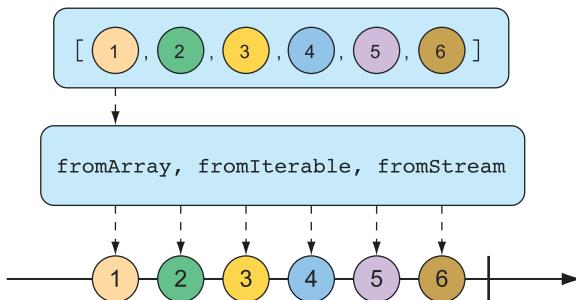


Рис. 11.3 Экземпляр Flux можно создать из массива, экземпляра Iterable или Stream

Чтобы создать экземпляр Flux из массива, нужно вызвать статический метод `fromArray()` и передать ему исходный массив:

```

@Test
public void createAFluxFromArray() {
    String[] fruits = new String[] {
        "Apple", "Orange", "Grape", "Banana", "Strawberry" };

    Flux<String> fruitFlux = Flux.fromArray(fruits);

    StepVerifier.create(fruitFlux)
        .expectNext("Apple")
        .expectNext("Orange")
        .expectNext("Grape")
        .expectNext("Banana")
        .expectNext("Strawberry")
        .verifyComplete();
}
  
```

Исходный массив в этом примере содержит те же имена фруктов, которые мы использовали при создании Flux из списка объектов, поэтому Flux будет выдавать те же значения и мы можем использовать для проверки тот же StepVerifier, что и раньше.

Чтобы создать Flux из `java.util.List`, `java.util.Set` или любой другой реализации `java.lang.Iterable`, нужно передать эту коллекцию в вызов статического метода `fromIterable()`:

```

@Test
public void createAFlux_fromIterable() {
    List<String> fruitList = new ArrayList<>();
    fruitList.add("Apple");
  
```

```

fruitList.add("Orange");
fruitList.add("Grape");
fruitList.add("Banana");
fruitList.add("Strawberry");

Flux<String> fruitFlux = Flux.fromIterable(fruitList);

StepVerifier.create(fruitFlux)
    .expectNext("Apple")
    .expectNext("Orange")
    .expectNext("Grape")
    .expectNext("Banana")
    .expectNext("Strawberry")
    .verifyComplete();
}

```

А если потребуется использовать в качестве источника данных Java-поток Stream, то нужно вызывать метод `fromStream()`:

```

@Test
public void createAFlux_fromStream() {
    Stream<String> fruitStream =
        Stream.of("Apple", "Orange", "Grape", "Banana", "Strawberry");

    Flux<String> fruitFlux = Flux.fromStream(fruitStream);

    StepVerifier.create(fruitFlux)
        .expectNext("Apple")
        .expectNext("Orange")
        .expectNext("Grape")
        .expectNext("Banana")
        .expectNext("Strawberry")
        .verifyComplete();
}

```

И снова для проверки данных можно использовать тот же `StepVerifier`, что и прежде.

ГЕНЕРИРОВАНИЕ НОВЫХ ДАННЫХ В ПОТОКЕ FLUX

Иногда изначально нет никаких данных для передачи и нужно, чтобы Flux работал как простой счетчик, выдавая последовательность чисел. Чтобы создать счетчик на основе Flux, можно использовать статический метод `range()`. Диаграмма на рис. 11.4 показывает, как работает `range()`.

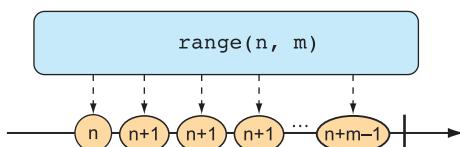


Рис. 11.4 Объект Flux, созданный с помощью метода `range()`, работает как простой счетчик

Следующий тест демонстрирует, как из Flux создать счетчик:

```
@Test
public void createAFlux_range() {
    Flux<Integer> intervalFlux =
        Flux.range(1, 5);

    StepVerifier.create(intervalFlux)
        .expectNext(1)
        .expectNext(2)
        .expectNext(3)
        .expectNext(4)
        .expectNext(5)
        .verifyComplete();
}
```

В этом примере создается счетчик Flux, генерирующий значения от 1 до 5. StepVerifier доказывает, что он действительно опубликует пять элементов – целые числа от 1 до 5.

Другой метод создания Flux, похожий на range(), – interval(). Подобно range(), метод interval() создает экземпляр Flux, который выдает последовательность возрастающих значений. Но особенность interval() в том, что вместо начального и конечного значений методу interval() передается продолжительность паузы перед выдачей очередного значения. На рис. 11.5 показана мраморная диаграмма, иллюстрирующая работу метода interval().

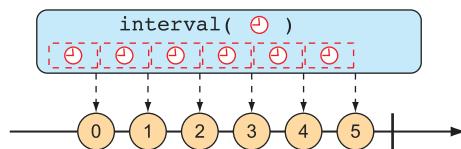


Рис. 11.5 Объект Flux, созданный с помощью метода interval(), выдает данные с указанной частотой

Например, чтобы создать Flux, генерирующий новое значение каждую секунду, можно вызвать статический метод interval(), как показано ниже:

```
@Test
public void createAFlux_interval() {
    Flux<Long> intervalFlux =
        Flux.interval(Duration.ofSeconds(1))
            .take(5);

    StepVerifier.create(intervalFlux)
        .expectNext(0L)
        .expectNext(1L)
        .expectNext(2L)
        .expectNext(3L)
        .expectNext(4L)
```

```

        .verifyComplete();
    }
}

```

Обратите внимание, что здесь Flux первым выдает значение 0 и затем увеличивает его в каждом следующем элементе. Кроме того, поскольку `interval()` не имеет параметра, определяющего максимальное значение, он может работать вечно. Поэтому здесь также использована операция `take()`, чтобы ограничить набор результатов первыми пятью элементами. Подробнее об операции `take()` мы поговорим в следующем разделе.

11.3.2 Комбинирование реактивных типов

Иногда случается так, что в программе появляются два реактивных типа, которые нужно как-то объединить. Иногда также может понадобиться разделить Flux на два или более реактивных типа. В этом разделе мы познакомимся с операциями, которые объединяют и разделяют экземпляры реактивных типов Flux и Mono.

Объединение реактивных типов

Предположим, что у нас есть два потока Flux и нужно создать один поток Flux, который будет обрабатывать и передавать данные, поступающие из любого из вышестоящих потоков Flux. Объединить два потока Flux в один можно с помощью операции `mergeWith()`, как показано на мраморной диаграмме на рис. 11.6.

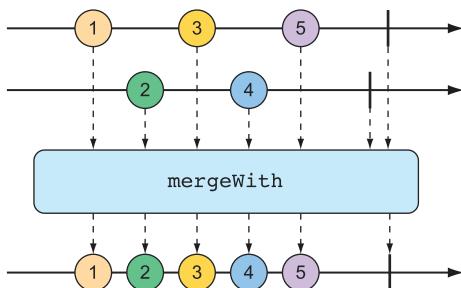


Рис. 11.6 Слияние двух потоков Flux путем поочередного объединения их элементов в один поток Flux

Например, предположим, что у нас есть два потока Flux: один производит имена персонажей телешоу и фильмов, а второй – названия любимых блюд этих персонажей. Следующий тестовый метод показывает, как можно объединить два объекта Flux с помощью метода `mergeWith()`:

```

@Test
public void mergeFluxes() {
    Flux<String> characterFlux = Flux

```

```
.just("Garfield", "Kojak", "Barbossa")
.delayElements(Duration.ofMillis(500));
Flux<String> foodFlux = Flux
.just("Lasagna", "Lollipops", "Apples")
.delaySubscription(Duration.ofMillis(250))
.delayElements(Duration.ofMillis(500));

Flux<String> mergedFlux = characterFlux.mergeWith(foodFlux);

StepVerifier.create(mergedFlux)
.expectNext("Garfield")
.expectNext("Lasagna")
.expectNext("Kojak")
.expectNext("Lollipops")
.expectNext("Barbossa")
.expectNext("Apples")
.verifyComplete();
}
```

Обычно Flux публикует данные почти мгновенно. Поэтому здесь к обоим созданным потокам Flux применена операция `delayElements()`, чтобы немного замедлить их и отправлять элементы с перерывами в 500 мс. Кроме того, чтобы foodFlux начинал посыпать свои элементы только после того, как это начнет делать characterFlux, к foodFlux применена операция `delaySubscription()`, вследствие чего отправка данных этим потоком начнется только через 250 мс после оформления подписки.

В результате объединения двух объектов Flux создается новый объединенный поток Flux. Когда StepVerifier подписывается на этот объединенный поток, тот в свою очередь подписывается на два исходных потока Flux, и передача данных начинается.

Порядок следования элементов в объединенном потоке соответствует порядку их отправки источниками. Поскольку оба объекта Flux настроены на отправку с постоянной скоростью, элементы в объединенном потоке будут чередоваться, т. е. следовать в таком порядке: персонаж, его любимое блюдо, следующий персонаж и его любимое блюдо и т. д. Если величину задержки в любом исходном потоке Flux изменить, то в объединенном потоке можно увидеть два персонажа или два блюда, следующих друг за другом.

Функция `mergeWith()` не может гарантировать идеального взаимодействия между источниками, поэтому вместо нее можно использовать операцию `zip()`. Объединяя два потока Flux, эта операция создает новый объект Flux, посылающий кортежи с элементами, по одному из каждого исходного потока. На рис. 11.7 показано, как происходит такое объединение.

Следующий тестовый метод иллюстрирует объединение потоков characterFlux и foodFlux с помощью операции `zip()`:

```
@Test
public void zipFluxes() {
```

```

Flux<String> characterFlux = Flux
    .just("Garfield", "Kojak", "Barbossa");
Flux<String> foodFlux = Flux
    .just("Lasagna", "Lollipops", "Apples");

Flux<Tuple2<String, String>> zippedFlux =
    Flux.zip(characterFlux, foodFlux);

StepVerifier.create(zippedFlux)
    .expectNextMatches(p ->
        p.getT1().equals("Garfield") &&
        p.getT2().equals("Lasagna"))
    .expectNextMatches(p ->
        p.getT1().equals("Kojak") &&
        p.getT2().equals("Lollipops"))
    .expectNextMatches(p ->
        p.getT1().equals("Barbossa") &&
        p.getT2().equals("Apples"))
    .verifyComplete();
}

```

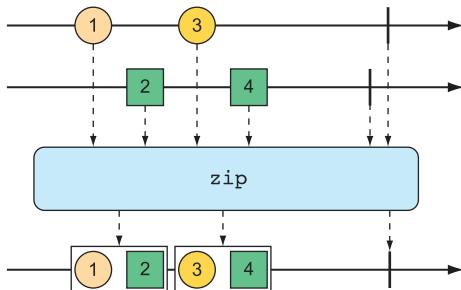


Рис. 11.7 Слияние двух потоков Flux путем объединения их элементов в кортежи

Обратите внимание, что в отличие от `mergeWith()` операция `zip()` является статической операцией. Созданный ею поток `Flux` обеспечивает идеальное соответствие между персонажами и их любимыми блюдами.

Каждый элемент в потоке, созданном операцией `zip()`, является экземпляром `Tuple2` (объектом-контейнером, содержащим два других объекта), содержащим элементы из обоих исходных потоков `Flux` в том порядке, в каком они были опубликованы.

Если вы предпочитаете вместо `Tuple2` использовать какой-то другой тип, то просто передайте в операцию `zip()` свою функцию для создания нужных вам объектов из двух элементов на входе (как показано на диаграмме на рис. 11.8).

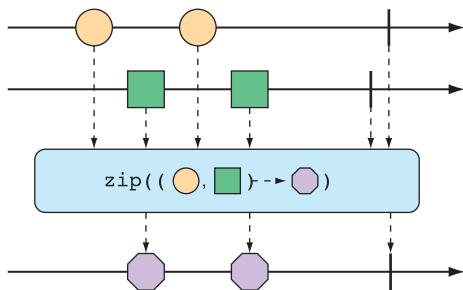


Рис. 11.8 Альтернативный способ использования операции `zip()`, который дает в результате поток сообщений, сконструированных из пар элементов, опубликованных исходными потоками

Например, следующий тестовый метод объединяет `characterFlux` и `foodFlux`, давая в результате поток объектов `String`:

```
@Test
public void zipFluxesToObject() {

    Flux<String> characterFlux = Flux
        .just("Garfield", "Kojak", "Barbossa");
    Flux<String> foodFlux = Flux
        .just("Lasagna", "Lollipops", "Apples");

    Flux<String> zippedFlux =
        Flux.zip(characterFlux, foodFlux, (c, f) -> c + " eats " + f);

    StepVerifier.create(zippedFlux)
        .expectNext("Garfield eats Lasagna")
        .expectNext("Kojak eats Lollipops")
        .expectNext("Barbossa eats Apples")
        .verifyComplete();
}
```

Функция, что передается в вызов `zip()` (в этом примере – лямбда-выражение), просто объединяет два элемента в объект некоторого типа, который будет опубликован объединенным потоком `Flux`.

ВЫБОР РЕАКТИВНОГО ПОТОКА, ПЕРВЫМ ОТПРАВИВШЕГО СОБЫТИЕ

Предположим, у нас есть два объекта `Flux` и нам нужно не объединить их, а просто выбрать поток, который первым начнет публиковать свои элементы. Операция `firstWithSignal()`, как показано на рис. 11.9, выбирает один из двух объектов `Flux`, который первым опубликовал значение, и пересыпает это значение дальше.

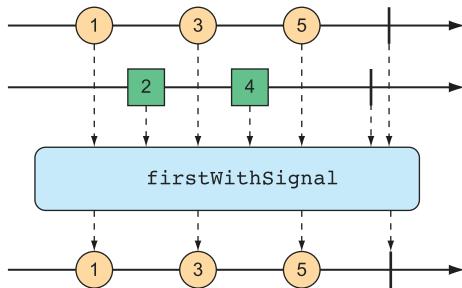


Рис. 11.9 Операция `firstWithSignal()` выбирает поток, первым опубликовавший сообщение, а затем пересыпает только сообщения из этого потока

Следующий тестовый метод создает два потока – `fastFlux` и `slowFlux` (поток `slowFlux` опубликует свой первый элемент с задержкой 100 мс после подписки). Затем, используя `firstWithSignal()`, он создает новый поток `Flux`, который будет публиковать значения только из первого исходного `Flux`.

```
@Test
public void firstWithSignalFlux() {
    Flux<String> slowFlux = Flux.just("tortoise", "snail", "sloth")
        .delaySubscription(Duration.ofMillis(100));
    Flux<String> fastFlux = Flux.just("hare", "cheetah", "squirrel");

    Flux<String> firstFlux = Flux.firstWithSignal(slowFlux, fastFlux);

    StepVerifier.create(firstFlux)
        .expectNext("hare")
        .expectNext("cheetah")
        .expectNext("squirrel")
        .verifyComplete();
}
```

В этом случае, поскольку `slowFlux` начинает публиковать свои значения с задержкой 100 мс, вновь созданный поток `Flux` просто проигнорирует его и будет публиковать только значения из `fastFlux`.

11.3.3 Преобразование и фильтрация реактивных потоков

Иногда бывает желательно фильтровать и/или изменять значения, проходящие через поток. В этом разделе мы рассмотрим операции, которые преобразуют и фильтруют данные в реактивном потоке.

Фильтрация данных в реактивных потоках

Самый простой способ отфильтровать нежелательные данные в потоке `Flux` – просто игнорировать несколько первых элементов. Именно это делает операция `skip()`, показанная на рис. 11.10.

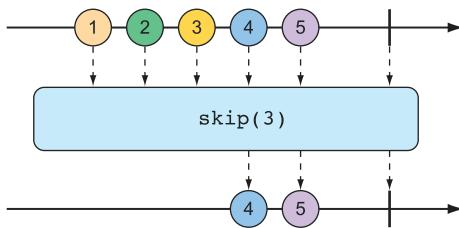


Рис. 11.10 Операция `skip()` пропускает указанное количество сообщений, после чего разрешает передачу оставшихся сообщений

Получив поток с несколькими элементами, операция `skip()` создаст новый поток, отбросит указанное количество элементов и затем начнет пересыпать остальные. Следующий тестовый метод показывает, как использовать `skip()`:

```
@Test
public void skipAFew() {
    Flux<String> countFlux = Flux.just(
        "one", "two", "skip a few", "ninety nine", "one hundred")
        .skip(3);

    StepVerifier.create(countFlux)
        .expectNext("ninety nine", "one hundred")
        .verifyComplete();
}
```

В этом случае у нас есть поток `Flux` с пятью элементами `String`. Вызов `skip(3)` создаст новый поток `Flux`, который отбросит первые три элемента и опубликует последние два.

Иногда бывает нужно отбросить не определенное количество элементов, а продолжать отбрасывать их, пока не пройдет некоторое время. Альтернативная форма операции `skip()`, показанная на рис. 11.11, создает поток, который ожидает, пока не пройдет определенное время, и только потом начинает публиковать элементы из исходного потока.

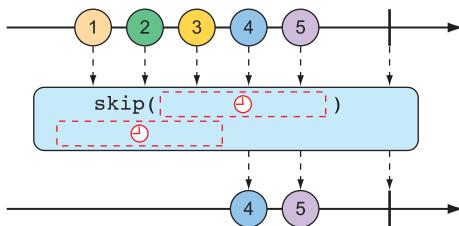


Рис. 11.11 Альтернативная форма операции `skip()` ожидает, пока не пройдет некоторое время, прежде чем начать передавать сообщения во вновь созданный поток

Следующий тестовый метод использует `skip()` для создания потока `Flux`, который ждет 4 секунды и затем начинает передавать значения

из исходного потока. Поскольку в этом примере новый поток создается из потока, публикующего элементы с 1-секундной задержкой (с помощью `delayElements()`), то он опубликует только два последних элемента.

```
@Test
public void skipAFewSeconds() {

    Flux<String> countFlux = Flux.just(
        "one", "two", "skip a few", "ninety nine", "one hundred")
        .delayElements(Duration.ofSeconds(1))
        .skip(Duration.ofSeconds(4));

    StepVerifier.create(countFlux)
        .expectNext("ninety nine", "one hundred")
        .verifyComplete();
}
```

Вы уже видели пример операции `take()`, но в свете операции `skip()` можно рассматривать `take()` как ее противоположность. Операция `skip()` отбрасывает несколько первых элементов, а `take()`, наоборот, передает только несколько первых элементов (как показано на диаграмме на рис. 11.12):

```
@Test
public void take() {

    Flux<String> nationalParkFlux = Flux.just(
        "Yellowstone", "Yosemite", "Grand Canyon", "Zion", "Acadia")
        .take(3);

    StepVerifier.create(nationalParkFlux)
        .expectNext("Yellowstone", "Yosemite", "Grand Canyon")
        .verifyComplete();
}
```

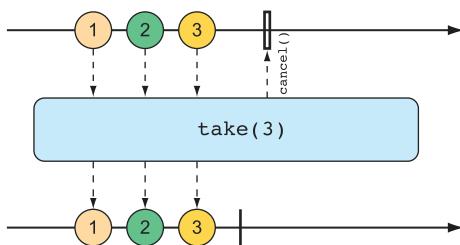


Рис. 11.12 Операция `take()` передает только первые несколько сообщений из исходного потока `Flux`, а затем отменяет подписку

Подобно `skip()`, операция `take()` тоже имеет альтернативную форму, основанную на длительности, а не на количестве элементов. Она будет передавать столько элементов, сколько поступит из исходного

потока, пока не истечет заданный интервал времени, после чего работа потока завершится. Это показано на рис. 11.13.

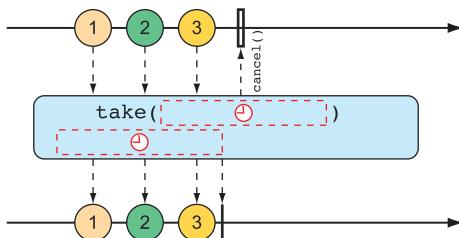


Рис. 11.13 Альтернативная форма операции take() передает сообщения в возвращаемый поток, пока не пройдет заданное время

В следующем тестовом методе используется альтернативная форма take(), передающая столько элементов, сколько успеет пройти в течение первых 3,5 секунды после подписки:

```

@Test
public void takeForAwhile() {
    Flux<String> nationalParkFlux = Flux.just(
        "Yellowstone", "Yosemite", "Grand Canyon", "Zion", "Grand Teton")
        .delayElements(Duration.ofSeconds(1))
        .take(Duration.ofMillis(3500));

    StepVerifier.create(nationalParkFlux)
        .expectNext("Yellowstone", "Yosemite", "Grand Canyon")
        .verifyComplete();
}

```

Операции skip() и take() можно рассматривать как операции фильтрации, критерии которых основаны на количестве или длительности. Более универсальный способ фильтрации значений в потоке Flux предлагает операция filter().

Операция filter() принимает предикат, с помощью которого будет определяться возможность передачи каждого конкретного элемента через поток. Диаграмма на рис. 11.14 показывает, как работает filter().

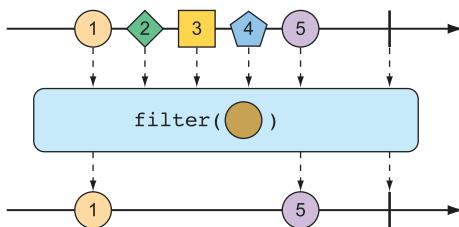


Рис. 11.14 Элементы в потоке Flux можно фильтровать и передавать только сообщения, соответствующие заданному предикату

Следующий тестовый метод иллюстрирует применение операции `filter()`:

```
@Test+
public void filter() {
    Flux<String> nationalParkFlux = Flux.just(
        "Yellowstone", "Yosemite", "Grand Canyon", "Zion", "Grand Teton")
        .filter(np -> !np.contains(" "));
    StepVerifier.create(nationalParkFlux)
        .expectNext("Yellowstone", "Yosemite", "Zion")
        .verifyComplete();
}
```

Здесь метод `filter()` получает предикат в виде лямбда-выражения, который пропускает только строковые значения без пробелов. То есть `"Grand Canyon"` и `"Grand Teton"` не попадут в результирующий поток.

Иногда бывает нужно отфильтровать элементы, которые уже были получены. Операция `distinct()` (рис. 11.15) пропустит только те элементы из исходного потока, которые еще не были опубликованы.

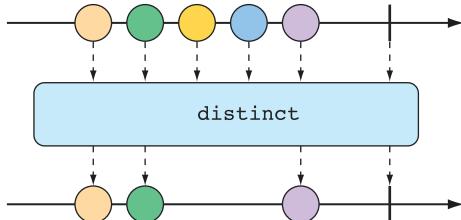


Рис. 11.15 Операция `distinct()` отфильтрует все повторяющиеся сообщения

В следующем тестовом методе в результирующий поток будут переданы лишь уникальные строковые значения:

```
@Test
public void distinct() {
    Flux<String> animalFlux = Flux.just(
        "dog", "cat", "bird", "dog", "bird", "anteater")
        .distinct();
    StepVerifier.create(animalFlux)
        .expectNext("dog", "cat", "bird", "anteater")
        .verifyComplete();
}
```

Исходный поток `Flux` дважды публикует сообщения `"dog"` и `"bird"`, но поток `Flux`, созданный операцией `distinct()`, опубликует их только один раз.

ОТБРАЖЕНИЕ РЕАКТИВНЫХ ДАННЫХ

Одной из наиболее распространенных операций, которые часто используются с потоками Flux и Mono, является преобразование опубликованных элементов в какую-либо другую форму или тип. Для этой цели Reactor предлагает операции `map()` и `flatMap()`.

Операция `map()` создает поток, который просто преобразует все получаемые элементы с использованием заданной функции и затем публикует результаты. На рис. 11.16 показано, как работает операция `map()`.

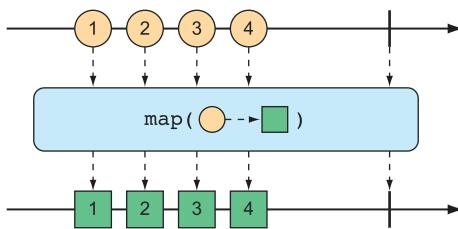


Рис. 11.16 Операция `map()` преобразует входящие сообщения и публикует их в результирующем потоке

В следующем тестовом методе значения типа `String` из исходного потока Flux с именами баскетболистов преобразуются (отображаются) в новом потоке Flux в объекты `Player`:

```
@Test
public void map() {
    Flux<Player> playerFlux = Flux
        .just("Michael Jordan", "Scottie Pippen", "Steve Kerr")
        .map(n -> {
            String[] split = n.split("\\s");
            return new Player(split[0], split[1]);
        });

    StepVerifier.create(playerFlux)
        .expectNext(new Player("Michael", "Jordan"))
        .expectNext(new Player("Scottie", "Pippen"))
        .expectNext(new Player("Steve", "Kerr"))
        .verifyComplete();
}

@Data
private static class Player {
    private final String firstName;
    private final String lastName;
}
```

Операция `map()` принимает функцию (в этом примере – лямбда-выражение), разбивает входную строку по пробелам и на основе по-

лученного массива создает объект `Player`. Если поток, созданный вызовом `just()`, содержит объекты `String`, то поток, созданный вызовом `map()`, будет содержать объекты `Player`.

Важно помнить, что `map()` выполняет преобразование синхронно, по мере публикации элементов исходным потоком `Flux`. Если преобразование должно выполняться асинхронно, то используйте операцию `flatMap()`.

Для успешного применения операции `flatMap()` требуется некоторая практика. Как показано на рис. 11.17, вместо простого отображения одного объекта в другой, как в случае с `map()`, операция `flatMap()` отображает каждый объект в новый поток `Mono` или `Flux`. Затем получившиеся потоки `Mono` или `Flux` объединяются в результирующий поток `Flux`. При использовании вместе с `subscribeOn()` операция `flatMap()` позволяет в полной мере раскрыть всю асинхронную мощь типов `Reactor`.

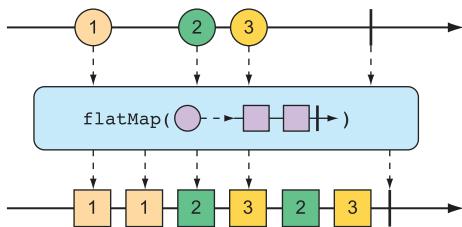


Рис. 11.17 Операция `flatMap()` преобразует исходные элементы и публикует их в промежуточные потоки `Flux`, что позволяет выполнять преобразования асинхронно

Следующий тестовый метод демонстрирует использование `flatMap()` и `subscribeOn()`:

```

@Test
public void flatMap() {
    Flux<Player> playerFlux = Flux
        .just("Michael Jordan", "Scottie Pippen", "Steve Kerr")
        .flatMap(n -> Mono.just(n)
            .map(p -> {
                String[] split = p.split("\\s");
                return new Player(split[0], split[1]);
            })
            .subscribeOn(Schedulers.parallel())
        );
    List<Player> playerList = Arrays.asList(
        new Player("Michael", "Jordan"),
        new Player("Scottie", "Pippen"),
        new Player("Steve", "Kerr"));
    StepVerifier.create(playerFlux)

```

```

    .expectNextMatches(p -> playerList.contains(p))
    .expectNextMatches(p -> playerList.contains(p))
    .expectNextMatches(p -> playerList.contains(p))
    .verifyComplete();
}

```

Обратите внимание, что `flatMap()` принимает функцию (в этом примере – лямбда-выражение), которая преобразует входную строку в поток Mono типа `String`. Затем к потоку Mono применяется операция `map()`, преобразующая строку в объект `Player`. После преобразования строк в объекты `Player` во всех промежуточных потоках полученные результаты публикуются в одном общем потоке, который возвращает `flatMap()`.

Если остановиться на этом, то полученный поток будет содержать объекты `Player`, созданные синхронно и в том же порядке, что и в примере с `map()`. Но в нашем примере к потокам Mono применяется еще одна операция – `subscribeOn()`, чтобы указать, что каждая подписка должна производиться в параллельном потоке выполнения. Как следствие операции отображения нескольких входных объектов `String` могут выполняться асинхронно и параллельно.

Несмотря на сходство имен `subscribeOn()` и `subscribe()`, это совершенно разные операции. Операция `subscribe()` – это глагол, означающий «подписаться на реактивный поток» и фактически запускающий его, а операция `subscribeOn()` является скорее определением, описывающим, как должна осуществляться конкурентная подписка. Reactor не навязывает какой-то определенной модели конкурентного выполнения; и именно через `subscribeOn()` можно указать модель, передав один из статических методов класса `Schedulers`. В этом примере указан метод `parallel()`, который использует рабочие потоки из фиксированного пула (размер этого пула соответствует количеству ядер процессора). Но вообще класс `Schedulers` поддерживает несколько моделей конкурентного выполнения, в том числе перечисленные в табл. 11.1.

Таблица 11.1 Модели конкурентного выполнения, поддерживаемые классом `Schedulers`

Метод в классе <code>Schedulers</code>	Описание
<code>.immediate()</code>	Выполняет подписку в текущем потоке
<code>.single()</code>	Выполняет подписку в одном многократно используемом потоке. Повторно использует один и тот же поток для всех вызовов
<code>.newSingle()</code>	Выполняет каждую подписку в отдельном потоке выполнения
<code>.elastic()</code>	Выполняет подписку в рабочем потоке, извлеченном из неограниченного расширяемого пула. Новые рабочие потоки создаются в этом пуле по мере необходимости, а простаивающие – удаляются (по умолчанию через 60 с)
<code>.parallel()</code>	Выполняет подписку в рабочем потоке, извлеченном из пула фиксированного размера, соответствующего количеству ядер процессора

Основное преимущество `flatMap()` и `subscribeOn()` заключается в возможности увеличить пропускную способность потока, распреде-

лив работу между несколькими параллельными потоками. Но, так как работа выполняется в параллельных потоках, порядок их завершения не гарантируется, поэтому невозможно заранее предсказать, в каком порядке будут опубликованы элементы в результирующем потоке. Таким образом, `StepVerifier` может проверить только присутствие ожидаемых элементов в потоке и их общее количество.

БУФЕРИЗАЦИЯ ДАННЫХ В РЕАКТИВНОМ ПОТОКЕ

В процессе обработки данных, проходящих через поток `Flux`, бывает полезно разбить поток на фрагменты определенного размера. В этом может помочь операция `buffer()`, показанная на рис. 11.18.

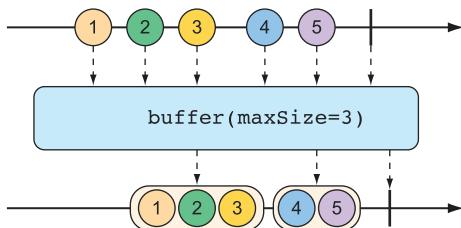


Рис. 11.18 Операция `buffer()` возвращает поток списков заданного максимального размера с элементами из исходного потока

Имея поток `Flux` значений типа `String`, каждое из которых содержит название фрукта, можно создать новый поток `Flux` списков `List`, каждый из которых содержит не более указанного количества элементов:

```

@Test
public void buffer() {
    Flux<String> fruitFlux = Flux.just(
        "apple", "orange", "banana", "kiwi", "strawberry");
    Flux<List<String>> bufferedFlux = fruitFlux.buffer(3);
    StepVerifier
        .create(bufferedFlux)
        .expectNext(Arrays.asList("apple", "orange", "banana"))
        .expectNext(Arrays.asList("kiwi", "strawberry"))
        .verifyComplete();
}
  
```

В этом случае элементы `String` объединяются в новые коллекции `List`, содержащие до трех элементов каждая. То есть исходный поток `Flux`, выдающий пять значений `String`, будет преобразован в новый поток `Flux`, выдающий две коллекции `List`, одна из которых содержит три строки, а другая – две.

Но зачем это? Буферизация значений из реактивного потока `Flux` в нереактивные коллекции `List` кажется контрпродуктивной. Однако,

объединив `buffer()` и `flatMap()`, эти коллекции `List` можно обрабатывать параллельно, как показано ниже:

```
@Test
public void bufferAndFlatMap() throws Exception {
    Flux.just(
        "apple", "orange", "banana", "kiwi", "strawberry")
        .buffer(3)
        .flatMap(x ->
            Flux.fromIterable(x)
                .map(y -> y.toUpperCase())
                .subscribeOn(Schedulers.parallel())
                .log()
        ).subscribe();
}
```

В этом новом примере мы все так же буферизуем поток `Flux` с пятью значениями `String` и выдаем новый поток `Flux` списков `List`. Затем применяем `flatMap()` к этим спискам. Эта операция извлекает каждый список, создает новый поток из его элементов и применяет к нему операцию `map()`. Как следствие каждый список обрабатывается параллельно в отдельном потоке.

Чтобы убедиться, что все работает именно так, как описано, я применил операцию `log()` к каждому потоку. Операция `log()` просто регистрирует все события Reactive Streams, чтобы можно было увидеть, что происходит на самом деле. В результате в журнал были записаны следующие события (для краткости я удалил время из каждой записи):

```
[main] INFO reactor.Flux.SubscribeOn.1 -
    onSubscribe(FluxSubscribeOn.SubscribeOnSubscriber)
[main] INFO reactor.Flux.SubscribeOn.1 - request(32)
[main] INFO reactor.Flux.SubscribeOn.2 -
    onSubscribe(FluxSubscribeOn.SubscribeOnSubscriber)
[main] INFO reactor.Flux.SubscribeOn.2 - request(32)
[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onNext(APPLE)
[parallel-2] INFO reactor.Flux.SubscribeOn.2 - onNext(KIWI)
[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onNext(ORANGE)
[parallel-2] INFO reactor.Flux.SubscribeOn.2 - onNext(STRAWBERRY)
[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onNext(BANANA)
[parallel-1] INFO reactor.Flux.SubscribeOn.1 - onComplete()
[parallel-2] INFO reactor.Flux.SubscribeOn.2 - onComplete()
```

Записи в журнале ясно показывают, что строки в первом списке ("apple", "orange" и "banana") обрабатываются в потоке `parallel-1`, а строки во втором списке ("kiwi" и "strawberry") обрабатываются в потоке `parallel-2`. Строки из разных списков в журнале перемешаны между собой, отсюда можно сделать вывод, что списки обрабатывались параллельно.

Если по какой-то причине вам понадобится собрать в список все, что выдает поток `Flux`, то используйте `buffer()` без аргументов:

```
Flux<List<String>> bufferedFlux = fruitFlux.buffer();
```

В результате будет создан новый поток Flux, создающий список со всеми элементами, опубликованными исходным потоком. Того же результата можно добиться с помощью операции `collectList()`, показанной на диаграмме на рис. 11.19.

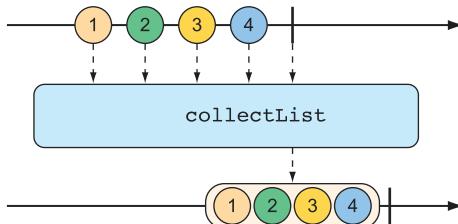


Рис. 11.19 Операция `collectList()` создает поток Mono со списком всех сообщений, отправленных исходным потоком Flux

Вместо создания потока Flux, публикующего список, операция `collectList()` создает поток Mono с этим списком. Следующий метод тестирования показывает, как его можно использовать:

```
@Test
public void collectList() {
    Flux<String> fruitFlux = Flux.just(
        "apple", "orange", "banana", "kiwi", "strawberry");

    Mono<List<String>> fruitListMono = fruitFlux.collectList();

    StepVerifier
        .create(fruitListMono)
        .expectNext(NSArray.asList(
            "apple", "orange", "banana", "kiwi", "strawberry"))
        .verifyComplete();
}
```

Еще более интересный способ объединения элементов, генерируемых потоком, – собрать их в ассоциативный массив. Как показано на рис. 11.20, операция `collectMap()` создает поток Mono, который публикует массив типа Map, заполненный элементами, ключи которых вычисляются заданной функцией.

Следующий тестовый метод иллюстрирует применение `collectMap()`:

```
@Test
public void collectMap() {
    Flux<String> animalFlux = Flux.just(
        "aardvark", "elephant", "koala", "eagle", "kangaroo");
```

```

Mono<Map<Character, String>> animalMapMono =
    animalFlux.collectMap(a -> a.charAt(0));

StepVerifier
    .create(animalMapMono)
    .expectNextMatches(map -> {
        return
            map.size() == 3 &&
            map.get('a').equals("aardvark") &&
            map.get('e').equals("eagle") &&
            map.get('k').equals("kangaroo");
    })
    .verifyComplete();
}

```

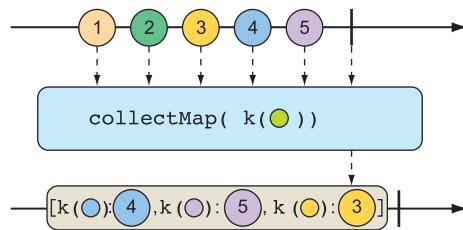


Рис. 11.20 Результатом операции `collectMap()` является поток Mono с ассоциативным массивом Map, содержащим сообщения, полученные из исходного потока Flux, где ключи вычисляются на основе некоторых характеристик сообщений

Исходный поток Flux выдает несколько названий животных. К этому потоку применяется операция `collectMap()`, создающая новый поток Mono с ассоциативным массивом Map, в котором ключами служат первые буквы названий животных, а значениями – сами названия. Если в потоке окажутся два названия, начинающиеся с одной и той же буквы (например, *elephant* (слон) и *eagle* (орел) или *koala* (коала) и *kangaroo* (кенгуру)), то последняя из них затрет все предшествующие.

11.3.4 Выполнение логических операций с реактивными типами

Иногда достаточно проверить соответствие элементов, опубликованных потоком Mono или Flux, некоторым критериям. Этую логику реализуют операции `all()` и `any()`. На рис. 11.21 и 11.22 показано, как они работают.

Допустим, что нам потребовалось выяснить, все ли строки в потоке Flux содержат букву *a* или *k*. Следующий тестовый метод показывает, как проверить это условие с помощью `all()`:

```

@Test
public void all() {

```

```

Flux<String> animalFlux = Flux.just(
    "aardvark", "elephant", "koala", "eagle", "kangaroo");

Mono<Boolean> hasAMono = animalFlux.all(a -> a.contains("a"));
StepVerifier.create(hasAMono)
    .expectNext(true)
    .verifyComplete();

Mono<Boolean> hasKMono = animalFlux.all(a -> a.contains("k"));
StepVerifier.create(hasKMono)
    .expectNext(false)
    .verifyComplete();
}

```

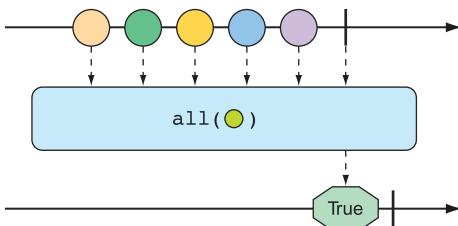


Рис. 11.21 С помощью операции all() можно проверить, все ли сообщения в потоке Flux удовлетворяют некоторому условию

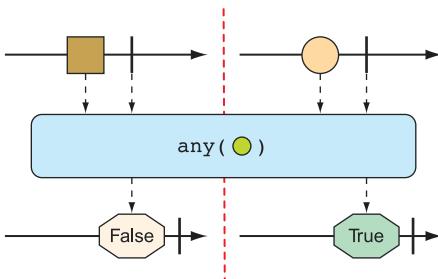


Рис. 11.22 С помощью операции any() можно проверить, имеется ли в потоке Flux хотя бы одно сообщение, удовлетворяющее некоторому условию

В первом вызове StepVerifier проверяется наличие буквы *a*. Операция all применяется к исходному потоку Flux и возвращает поток Mono с элементом типа Boolean. В этом случае все названия животных содержат букву *a*, поэтому результирующий поток Mono выдает true. Но во втором вызове StepVerifier результирующий поток Mono выдаст false, потому что не все названия животных содержат букву *k*.

Иногда вместо проверки по принципу «все или ничего» достаточно убедиться, что хотя бы один элемент в потоке соответствует условию. В этом случае вам поможет операция any(). Следующий тестовый метод использует any() для проверки наличия в строках букв *t* и *z*:

```
@Test
public void any() {

    Flux<String> animalFlux = Flux.just(
        "aardvark", "elephant", "koala", "eagle", "kangaroo");

    Mono<Boolean> hasTMono = animalFlux.any(a -> a.contains("t"));
    StepVerifier.create(hasTMono)
        .expectNext(true)
        .verifyComplete();

    Mono<Boolean> hasZMono = animalFlux.any(a -> a.contains("z"));
    StepVerifier.create(hasZMono)
        .expectNext(false)
        .verifyComplete();
}
```

В первом вызове `StepVerifier` результирующий поток `Mono` выдаст `true`, потому что по крайней мере одно название животного содержит букву *t* (в частности, *elephant*). Во втором случае результирующий поток `Mono` выдаст `false`, потому что ни одно из названий не содержит буквы *z*.

Итоги

- Реактивное программирование предполагает создание конвейеров, через которые передаются данные.
- Спецификация Reactive Streams определяет четыре типа: `Publisher`, `Subscriber`, `Publisher`, `Subscriber`, `Subscription` и `Transformer` (последний является комбинацией `Publisher` и `Subscriber`).
- Project Reactor реализует библиотеку реактивных потоков Reactive Streams и определяет два основных типа потоков: `Flux` и `Mono`. Оба этих типа поддерживают сотни операций.
- Spring использует Reactor в реализациях реактивных контроллеров, репозиториев, клиентов REST и других средств поддержки реактивной инфраструктуры.

12 Разработка реактивных API

В этой главе рассматриваются следующие темы:

- Spring WebFlux;
- создание и тестирование реактивных контроллеров и клиентов;
- использование REST API;
- безопасность реактивных веб-приложений.

Теперь, получив представление о реактивном программировании и Project Reactor, мы готовы начать применять новые методы в своих приложениях Spring. В этой главе мы вернемся к некоторым контроллерам, которые написали в главе 7, и воспользуемся преимуществами модели реактивного программирования Spring.

В частности, мы исследуем реактивный веб-фреймворк Spring WebFlux. Как вы увидите сами, он очень похож на Spring MVC и для его использования вполне достаточно знаний о создании REST API в Spring, которые у вас уже есть.

12.1 Spring WebFlux

Типичные веб-фреймворки сервлетов, такие как Spring MVC, являются блокирующими и многопоточными по своей природе – для обслу-

живания каждого соединения они используют отдельный поток выполнения. При поступлении очередного запроса для его обработки из пула извлекается свободный рабочий поток. При этом поток, обслуживающий соединение, блокируется до тех пор, пока рабочий поток не уведомит его о завершении.

По этой причине блокирующие веб-фреймворки неспособны справиться с обработкой большого количества запросов. Задержка, связанная с извлечением рабочего потока из пула и возвратом его в пул, еще больше усугубляет ситуацию, потому что на это зачастую тратится больше времени, чем на обработку самого запроса. В некоторых случаях подобная организация работы вполне приемлема, и в действительности именно так разрабатывалось большинство веб-приложений на протяжении многих лет. Но времена меняются.

Количество клиентов таких веб-приложений значительно возросло. Если раньше люди лишь время от времени просматривали веб-сайты, то теперь многие пользуются информацией из интернета на постоянной основе и используют приложения, зависящие от HTTP API. Более того, в наши дни бурно развивается так называемый *интернет вещей* (Internet of Things, IoT), посредством которого автомобили, реактивные двигатели и другие нетрадиционные клиенты постоянно обмениваются данными с веб-API. С ростом числа клиентов, использующих веб-приложения, масштабируемость приобрела особую важность.

Асинхронные веб-фреймворки, напротив, обеспечивают более высокую масштабируемость с меньшим количеством потоков – обычно по одному на ядро процессора. Применяя прием, известный как *цикл обработки событий* (рис. 12.1), эти фреймворки способны обрабатывать множество запросов в одном потоке выполнения, что существенно уменьшает расходы вычислительных ресурсов на обслуживание одного соединения.

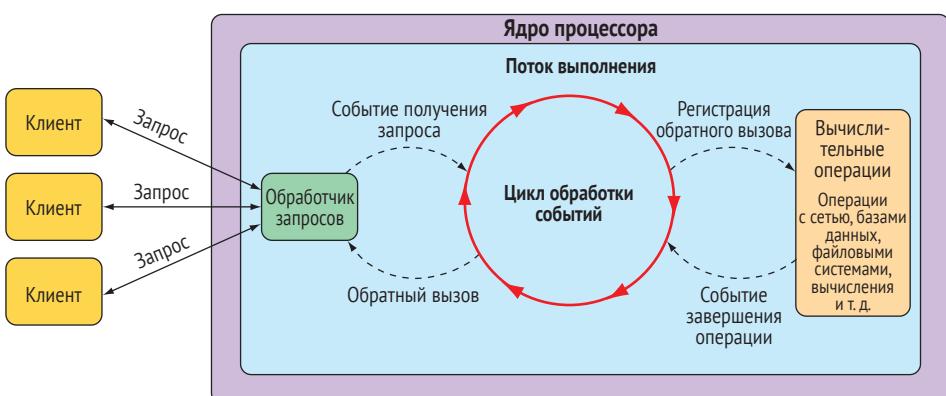


Рис. 12.1 Асинхронные веб-фреймворки используют цикл обработки событий для обслуживания большего количества запросов меньшим количеством потоков

В цикле событий все, включая запросы и обратные вызовы с результатами вычислений, операций с базами данных и сетью, обрабатывается как события. Когда требуется выполнить продолжительную операцию, цикл событий регистрирует обратный вызов для нее и запускает ее в параллельном потоке выполнения, а сам переходит к обработке других событий.

Когда операция завершается, она посыпает событие, которое будет обработано циклом событий. В результате асинхронные веб-фреймворки лучше адаптируются для обработки большого количества запросов с меньшим количеством потоков, что в свою очередь снижает накладные расходы на управление потоками.

Spring предлагает неблокирующий асинхронный веб-фреймворк, основанный на Project Reactor, который способен удовлетворить самые строгие требования к масштабируемости веб-приложений и API – Spring WebFlux. Давайте взглянем на него поближе.

12.1.1 Введение в Spring WebFlux

Когда команда разработчиков Spring обдумывала, как добавить модель реактивного программирования в веб-слой, они быстро пришли к выводу, что сделать это будет очень сложно без полной реорганизации Spring MVC. Во фреймворк придется добавить новые ветви кода, выбирающие между обычной и реактивной обработками запросов. По сути, в результате получится два веб-фреймворка в одном с множеством операторов `if`, отделяющих реактивный код от нереактивного.

Поэтому вместо внедрения модели реактивного программирования в Spring MVC команда Spring решила создать отдельный реактивный веб-фреймворк, позаимствовав как можно больше из Spring MVC. В результате появился Spring WebFlux. На рис. 12.2 показан полный стек веб-разработки, доступный в Spring.

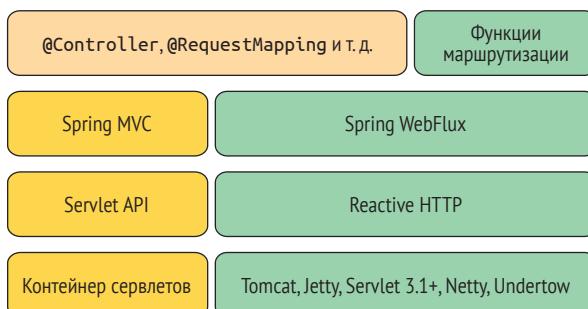


Рис. 12.2 Spring реализует поддержку реактивных веб-приложений в форме нового веб-фреймворка WebFlux, родственного фреймворку Spring MVC и использующего многие основные компоненты Spring MVC

Слева на рис. 12.2 показан стек Spring MVC, реализованный в версии 2.5 Spring Framework. Spring MVC (описанный в главах 2 и 7) дей-

ствует поверх Java Servlet API, который в свою очередь использует контейнер серверов (например, Tomcat).

Spring WebFlux (справа на рис. 12.2), напротив, не имеет привязки к Servlet API и основан на Reactive HTTP API – реактивной аппроксимации Servlet API, предоставляющей те же функциональные возможности. А поскольку Spring WebFlux не связан с Servlet API, ему не требуется контейнер серверов – он может работать в любом неблокирующем веб-контейнере, включая Netty, Undertow, Tomcat, Jetty, или любом контейнере, поддерживающем спецификацию Servlet 3.1 или выше.

Особого внимания на рис. 12.2 заслуживает верхний левый блок, представляющий компоненты, общие для Spring MVC и Spring WebFlux, в том числе и аннотации, используемые для определения контроллеров. Поскольку Spring MVC и Spring WebFlux используют одни и те же аннотации, Spring WebFlux во многом практически неотличим от Spring MVC.

Поле в правом верхнем углу представляет альтернативную модель программирования, которая использует для определения контроллеров парадигму функционального программирования вместо аннотаций. Подробнее о модели функционального веб-программирования в Spring мы поговорим в разделе 12.2.

Наиболее существенное различие между Spring MVC и Spring WebFlux заключается в зависимостях, которые нужно добавить в спецификацию сборки. В случае выбора Spring WebFlux нужно добавить начальную зависимость Spring Boot WebFlux вместо стандартной (такой как `spring-boot-starter-web`). В файле проекта `pom.xml` это выглядит так:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

ПРИМЕЧАНИЕ Как и большинство других начальных зависимостей Spring Boot, эту зависимость можно добавить в проект, установив флагок **Reactive Web** в приложении Initializr.

Интересным побочным эффектом использования WebFlux вместо Spring MVC является применение встроенного сервера по умолчанию Netty вместо Tomcat. Netty – один из немногих асинхронных серверов, управляемых событиями, идеально подходящий для обслуживания реактивных веб-фреймворков, таких как Spring WebFlux.

Помимо необходимости использовать другую начальную зависимость, методы контроллеров Spring WebFlux обычно принимают и возвращают реактивные типы, такие как `Mono` и `Flux`, вместо предметных типов и коллекций. Контроллеры Spring WebFlux также могут работать с типами RxJava, такими как `Observable`, `Single` и `Completable`.

Реактивный Spring MVC?

Контроллеры Spring WebFlux обычно возвращают Mono и Flux, но и Spring MVC тоже может использовать реактивные типы. И методы контроллера Spring MVC могут возвращать Mono или Flux.

Разница заключается в особенностях использования этих типов. Spring WebFlux – это по-настоящему реактивный веб-фреймворк, обрабатывающий запросы в цикле событий, а Spring MVC основан на сервлетах и для обработки запросов использует многопоточность.

Давайте запустим в работу Spring WebFlux, переписав некоторые контроллеры Taco Cloud API.

12.1.2 Создание реактивных контроллеров

Возможно, вы помните, что в главе 7 мы создали несколько контроллеров для Taco Cloud REST API. Эти контроллеры имели методы обработки запросов, принимающие и возвращающие данные предметных типов (таких как TacoOrder и Taco) или коллекций этих типов. Для напоминания ниже приводится фрагмент из TacoController, который мы написали еще в главе 7:

```
@RestController
@RequestMapping(path="/api/tacos",
    produces="application/json")
@CrossOrigin(origins="*")
public class TacoController {

    ...

    @GetMapping(params="recent")
    public Iterable<Taco> recentTacos() {
        PageRequest page = PageRequest.of(
            0, 12, Sort.by("createdAt").descending());
        return tacoRepo.findAll(page).getContent();
    }

    ...
}
```

Метод recentTacos() этого контроллера обрабатывает HTTP-запросы GET к конечной точке `/api/tacos?recent` и возвращает список последних созданных рецептов тако. В частности, он возвращает коллекцию Iterable с объектами Taco. Такое поведение обусловлено прежде всего тем, что он возвращает результат вызова метода findAll() репозитория или, точнее, метода getContent() объекта Page, возвращаемого методом findAll().

Это решение прекрасно работает, но Iterable не является реактивным типом. К нему нельзя применить реактивные операции, и фреймворк не сможет использовать его как реактивный тип для

распределения работы между несколькими потоками. А нам хотелось бы, чтобы `recentTacos()` возвращал `Flux<Taco>`.

Самое простое, но несколько ограниченное решение – переписать `recentTacos()` и преобразовать в нем `Iterable` в `Flux`. При этом появляется возможность избавиться от использования объекта `Page` и заменить его применением операции `take()` к потоку `Flux`:

```
@GetMapping(params="recent")
public Flux<Taco> recentTacos() {
    return Flux.fromIterable(tacoRepo.findAll()).take(12);
}
```

Вызов `Flux.fromIterable()` преобразует `Iterable<Taco>` в `Flux<Taco>`. И теперь, получив поток `Flux`, можно применить к нему операцию `take()`, чтобы ограничить количество элементов в возвращаемом потоке двенадцатью объектами `Taco`. Кроме того что код стал проще, он также оперирует реактивным потоком `Flux`, а не простой коллекцией `Iterable`.

Преобразование коллекции в реактивный поток уже дало нам некоторый выигрыш. Но было бы еще лучше, если бы поток `Flux` возвращался репозиторием, чтобы избежать дополнительного преобразования. В таком случае метод `recentTacos()` мог бы выглядеть так:

```
@GetMapping(params="recent")
public Flux<Taco> recentTacos() {
    return tacoRepo.findAll().take(12);
}
```

Этот код выглядит еще проще! В идеале реактивный контроллер должен быть вершиной реактивного стека, включающего другие контроллеры, репозитории, базы данных и любые промежуточные службы. Такой сквозной реактивный стек показан на рис. 12.3.

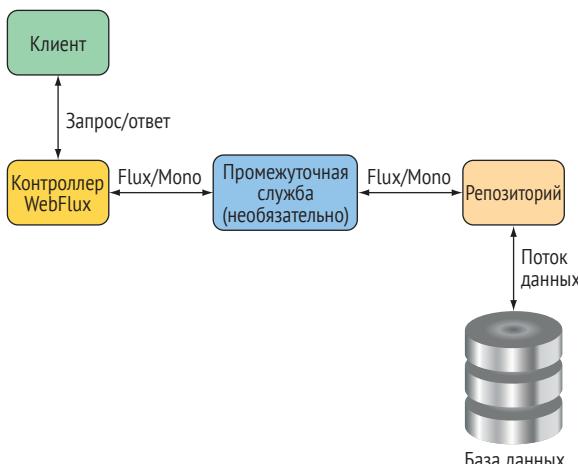


Рис. 12.3 Чтобы получить все преимущества реактивной модели, веб-фреймворк должен быть частью комплексного реактивного стека

Такой сквозной стек требует, чтобы репозиторий поддерживал возможность возврата потоков Flux вместо коллекций Iterable. Создание реактивных репозиториев мы рассмотрим в следующей главе, а пока лишь просто посмотрим, как может выглядеть реактивный TacoRepository:

```
package tacos.data;

import org.springframework.data.repository.reactive.ReactiveCrudRepository;
import tacos.Taco;

public interface TacoRepository
    extends ReactiveCrudRepository<Taco, Long> {
}
```

На этом этапе важно отметить, что помимо использования Flux вместо Iterable, а также способа получения этого потока Flux порядок определения реактивного контроллера WebFlux ничем не отличается от определения нереактивного контроллера Spring MVC. Оба снабжены аннотациями @RestController и @RequestMapping на уровне класса. Оба имеют методы обработки запросов с аннотациями @GetMapping на уровне метода. Фактически вся разница сводится к типу значения, возвращаемого методом-обработчиком.

Отмечу еще одно важное наблюдение: несмотря на то что метод контроллера получает поток Flux<Taco> из репозитория, он может вернуть его без вызова subscribe() – фреймворк сам вызовет его. Это означает, что, обработав запрос к конечной точке /api/tacos?recent, метод recentTacos() завершится еще до того, как данные будут извлечены из базы данных!

ВОЗВРАТ ОДИНОЧНЫХ ЗНАЧЕНИЙ

В качестве еще одного примера рассмотрим следующий метод tacoById() из TacoController. Вот как он был написан в главе 7:

```
@GetMapping("/{id}")
public Taco tacoById(@PathVariable("id") Long id) {
    Optional<Taco> optTaco = tacoRepo.findById(id);
    if (optTaco.isPresent()) {
        return optTaco.get();
    }
    return null;
}
```

Этот метод обрабатывает запросы GET к конечной точке /tacos/{id} и возвращает один объект Taco. Поскольку метод репозитория findById() возвращает значение типа Optional, нам также пришлось написать довольно неуклюжий код для его обработки. Но предположим на минуту, что findById() возвращает Mono<Taco> вместо Optional<Taco>. В этом случае метод tacoById() контроллера можно переписать так:

```
@GetMapping("/{id}")
public Mono<Taco> tacoById(@PathVariable("id") Long id) {
    return tacoRepo.findById(id);
}
```

Отлично! Так он выглядит намного проще. Однако более важно то, что, возвращая `Mono<Taco>` вместо Taco, мы позволяем фреймворку Spring WebFlux обрабатывать ответ в реактивной парадигме, а значит, наш API будет лучше масштабироваться при больших нагрузках.

Работа с типами RxJava

Типы из библиотеки Reactor, такие как `Flux` и `Mono`, являются естественным выбором при работе с Spring WebFlux, но с неменьшим успехом можно также использовать типы из библиотеки RxJava, такие как `Observable` и `Single`. Например, предположим, что между `TacoController` и репозиторием находится служба, работающая с типами RxJava. В этом случае метод `recentTacos()` можно реализовать так:

```
@GetMapping(params = "recent")
public Observable<Taco> recentTacos() {
    return tacoService.getRecentTacos();
}
```

Метод `tacoById()` тоже можно адаптировать для работы с типом `Single` из RxJava вместо `Mono`, как показано ниже:

```
@GetMapping("/{id}")
public Single<Taco> tacoById(@PathVariable("id") Long id) {
    return tacoService.lookupTaco(id);
}
```

Кроме того, методы контроллеров Spring WebFlux могут возвращать тип `CompletableFuture` из RxJava, эквивалентный типу `Mono<Void>` в Reactor. Также WebFlux может возвращать тип `Flowable` из RxJava вместо типа `Observable` или типа `Flux` из Reactor.

Реактивная обработка входных данных

До сих пор мы исследовали только способы возврата реактивных типов из методов контроллера. Но Spring WebFlux также позволяет принимать `Mono` или `Flux` в качестве входных данных. Для демонстрации рассмотрим исходную реализацию `postTaco()` из `TacoController`:

```
@PostMapping(consumes="application/json")
@ResponseStatus(HttpStatus.CREATED)
public Taco postTaco(@RequestBody Taco taco) {
    return tacoRepo.save(taco);
}
```

Наша первоначальная версия `postTaco()` не только возвращает, но и принимает простой объект `Taco`, соответствующий содержимому

в теле запроса. Это означает, что `postTaco()` нельзя вызвать, пока содержимое запроса не будет полностью проанализировано и использовано для создания экземпляра объекта Taco. Это также означает, что `postTaco()` не может вернуть управление, пока не завершится блокирующий вызов метода `save()` репозитория. Проще говоря, запрос блокируется дважды: на входе и затем внутри `postTaco()`. Но, применив парадигму реактивного программирования, как показано ниже, можно сделать метод обработки запросов полностью неблокирующем:

```
@PostMapping(consumes = "application/json")
@ResponseStatus(HttpStatus.CREATED)
public Mono<Taco> postTaco(@RequestBody Mono<Taco> tacoMono) {
    return tacoRepo.saveAll(tacoMono).next();
}
```

Здесь `postTaco()` принимает `Mono<Taco>` и вызывает метод репозитория `saveAll()`, который принимает любую реализацию Publisher из Reactive Streams, включая `Mono` и `Flux`. Метод `saveAll()` возвращает `Flux<Taco>`, но, начав с `Mono`, мы знаем, что `Flux` опубликует не более одного объекта Taco. Поэтому можно просто вызвать `next()`, чтобы получить `Mono<Taco>`, и вернуть его из `postTaco()`.

Принимая `Mono<Taco>` на входе, метод вызывается немедленно, не дожидаясь преобразования содержимого запроса в объект Taco. А поскольку репозиторий тоже является реактивным, он примет `Mono` и немедленно вернет `Flux<Taco>`, для которого мы вызываем `next()` и возвращаем полученный `Mono<Taco>`... еще до того, как запрос будет обработан!

В качестве альтернативы можно реализовать `postTaco()` так:

```
@PostMapping(consumes = "application/json")
@ResponseStatus(HttpStatus.CREATED)
public Mono<Taco> postTaco(@RequestBody Mono<Taco> tacoMono) {
    return tacoMono.flatMap(tacoRepo::save);
}
```

Это решение переворачивает все с ног на голову и `tacoMono` становится движущей силой. Объект Taco, содержащийся в `tacoMono`, передается методу `save()` репозитория через `flatMap()`, в результате чего возвращается новый `Mono<Taco>`.

Все эти способы хороши, и, вероятно, есть другие способы реализации `postTaco()`. Выбирайте тот, который вам кажется понятнее и лучше подходит для вашей ситуации. Spring WebFlux – это фантастическая альтернатива фреймворку Spring MVC, предлагающая возможность писать реактивные веб-приложения с использованием той же модели разработки, что и Spring MVC. Но у Spring есть еще один трюк. Давайте посмотрим, как создавать реактивные API, используя стиль функционального программирования Spring.

12.2 Определение обработчиков запросов в функциональном стиле

Модель программирования на основе аннотаций Spring MVC была реализована в версии Spring 2.5 и пользуется широкой популярностью. Однако она имеет несколько недостатков.

Во-первых, любое программирование с применением аннотаций предполагает разделение в том, что аннотация должна делать и как она должна это делать. Сами аннотации определяют, что делать, а как это делать, определяется в другом месте в коде фреймворка. Это деление усложняет модель программирования, когда речь заходит о корректировке или расширении поведения аннотаций, потому что для реализации таких изменений требуется менять код, внешний по отношению к аннотации. Более того, отладка такого кода чрезвычайно сложна из-за невозможности установки точки останова на аннотации.

Кроме того, с ростом популярности Spring на этот фреймворк мигрирует все больше разработчиков, которые имеют опыт программирования на других языках, с применением иных фреймворков и плохо знакомые с Spring. Они могут обнаружить, что приемы программирования с использованием аннотаций Spring MVC (и WebFlux) совершенно не похожи на известные им приемы. По этой причине WebFlux Spring предлагает альтернативу – функциональную модель программирования для определения реактивных API.

Эта новая модель программирования используется скорее как библиотека, а не как фреймворк. Она позволяет связывать запросы с обработчиками без аннотаций. Разработка API с применением модели функционального программирования Spring предполагает использование следующих четырех основных типов:

- *RequestPredicate* – объявляет типы обрабатываемых запросов;
- *RouterFunction* – определяет, как запрос должен передаваться в код обработчика;
- *ServerRequest* – представляет HTTP-запрос, включая заголовки и содержимое;
- *ServerResponse* – представляет HTTP-ответ, включая заголовки и содержимое.

Применение всех этих типов демонстрирует следующий простой пример Hello World:

```
package hello;

import static org.springframework.web
    .reactive.function.server.RequestPredicates.GET;
import static org.springframework.web
    .reactive.function.server.RouterFunctions.route;
import static org.springframework.web
    .reactive.function.server.ServerResponse.ok;
```

```

import static reactor.core.publisher.Mono.just;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.reactive.function.server.RouterFunction;

@Configuration
public class RouterFunctionConfig {

    @Bean
    public RouterFunction<?> helloRouterFunction() {
        return route(GET("/hello"),
            request -> ok().body(just("Hello World!"), String.class));
    }
}

```

Первое, на что следует обратить внимание: здесь мы статически импортируем несколько вспомогательных классов для создания вышеупомянутых функциональных типов. Мы также статически импортировали класс Mono, чтобы остальную часть кода было легче читать и понимать.

В нашем классе RouterFunctionConfig с аннотацией @Configuration определен единственный @Bean-метод, возвращающий RouterFunction<?>. Как уже упоминалось выше, RouterFunction определяет соответствие между одним или несколькими объектами Request-Predicate и функциями, которые будут обрабатывать соответствующие запросы.

Метод route() из RouterFunctions принимает два параметра: RequestPredicate и функцию для обработки соответствующих запросов. В этом случае метод GET() из пакета RequestPredicates объявляет экземпляр RequestPredicate, которому соответствуют HTTP-запросы GET к конечной точке /hello.

Сама функция-обработчик написана как лямбда-выражение, но точно так же можно передать ссылку на метод. Хотя это не видно по коду, но лямбда-обработчик принимает параметр типа ServerRequest и возвращает ServerResponse, используя метод ok() из ServerResponse и body() из BodyBuilder, возвращаемого методом ok(). Так создается ответ с кодом состояния HTTP 200 (OK) и с содержимым «Hello World!».

Согласно коду, метод helloRouterFunction() объявляет функцию RouterFunction, которая обрабатывает только один тип запроса. Но если вам понадобится обрабатывать запросы другого типа, то нет необходимости писать еще один метод с аннотацией @Bean, хотя это и не возбраняется. Достаточно будет вызвать andRoute(), чтобы объявить другой предикат RequestPredicate для определения соответствия между запросами и имеющейся функцией. Например, вот как можно добавить еще один обработчик запросов GET к конечной точке /bye:

```

@Bean
public RouterFunction<?> helloRouterFunction() {
    return route(GET("/hello"),

```

```
        request -> ok().body(just("Hello World!"), String.class))
            .andRoute(GET("/bye"),
        request -> ok().body(just("See ya!"), String.class));
}
```

Примеры «Hello World» отлично подходят, чтобы попробовать что-то новое. Но давайте немного расширим его и посмотрим, как использовать функциональную модель веб-программирования Spring для обработки запросов на примере более практического сценария.

Для демонстрации использования модели функционального программирования в реальном приложении давайте перепишем TacoController в функциональном стиле. Следующий класс конфигурации является функциональным аналогом TacoController:

```
package tacos.web.api;

import static org.springframework.web.reactive.function.server
    .RequestPredicates.GET;
import static org.springframework.web.reactive.function.server
    .RequestPredicates.POST;
import static org.springframework.web.reactive.function.server
    .RequestPredicates.queryParam;
import static org.springframework.web.reactive.function.server
    .RouterFunctions.route;

import java.net.URI;

import org.springframework.beans.factory.annotation.Autowired;
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.ServerRequest;
import org.springframework.web.reactive.function.server.ServerResponse;

import reactor.core.publisher.Mono;
import tacos.Taco;
import tacos.data.TacoRepository;

@Configuration
public class RouterFunctionConfig {

    @Autowired
    private TacoRepository tacoRepo;

    @Bean
    public RouterFunction<?> routerFunction() {
        return route(GET("/api/tacos").
            and(queryParam("recent", t->t != null )),
            this::recents)
            .andRoute(POST("/api/tacos"), this::postTaco);
    }

    public Mono<ServerResponse> recents(ServerRequest request) {
```

```

        return ServerResponse.ok()
            .body(tacoRepo.findAll().take(12), Taco.class);
    }

    public Mono<ServerResponse> postTaco(ServerRequest request) {
        return request.bodyToMono(Taco.class)
            .flatMap(taco -> tacoRepo.save(taco))
            .flatMap(savedTaco -> {
                return ServerResponse
                    .created(URI.create(
                        "http://localhost:8080/api/tacos/" +
                        savedTaco.getId()))
                    .body(savedTaco, Taco.class);
            });
    }
}

```

Как видите, метод `routerFunction()` объявляет компонент `RouterFunction<?>`, так же как в примере «Hello World». Но он отличается типами обрабатываемых запросов и особенностями их обработки. В данном случае `RouterFunction` будет обрабатывать запросы GET к конечной точке `/api/tacos?recent` и запросы POST к конечной точке `/api/tacos`.

Также обратите внимание, что обработчик передается методам `route()` и `addRoute()` по ссылкам. Лямбда-выражения удобно использовать, когда функция `RouterFunction` имеет относительно простое поведение. Но на практике часто лучше бывает выделить эту функциональность в отдельный метод (или даже в отдельный метод в отдельном классе), чтобы повысить удобочитаемость кода.

В нашем примере GET-запросы к конечной точке `/api/tacos?recent` будут обрабатываться методом `recent()`. Он использует внедренный компонент `TacoRepository` для получения потока `Flux<Taco>`, откуда извлекает 12 элементов. Затем мы заключаем `Flux<Taco>` в `Mono<ServerResponse>`, чтобы получить возможность отправить статус HTTP 200 (OK) вызовом `ok()` в `ServerResponse`. Важно понимать, что все 12 рецептов тако отправляются в одном ответе сервера, поэтому он возвращается в виде потока `Mono`, а не `Flux`. Внутри Spring клиенту по-прежнему будет передавать поток `Flux<Taco>`.

Запросы POST к конечной точке `/api/tacos` обрабатываются методом `postTaco()`, который извлекает `Mono<Taco>` из тела входящего запроса `ServerRequest`. Затем метод `postTaco()` применяет ряд операций `flatMap()` для сохранения полученного рецепта в `TacoRepository` и создания `ServerResponse` с кодом состояния HTTP 201 (CREATED) и сохраненным объектом `Taco` в теле ответа.

Операции `flatMap()` обеспечивают заключение результата отображения на каждом шаге в поток `Mono`, начиная с `Mono<Taco>` после первого `flatMap()` и заканчивая `Mono<ServerResponse>`, который возвращается из `postTaco()`.

12.3 Тестирование реактивных контроллеров

Когда дело доходит до тестирования реактивных контроллеров, фреймворк Spring не оставляет нас наедине с нашими проблемами. Для этой цели в Spring имеется `WebTestClient` – новая утилита тестирования, которая упрощает разработку тестов для реактивных контроллеров, написанных с помощью Spring WebFlux. Давайте посмотрим, как писать тесты с помощью `WebTestClient`, начнем с тестирования метода `recentTacos()` из `TacoController`, который мы написали в разделе 12.1.2.

12.3.1 Тестирование запросов GET

Прежде всего давайте убедимся, что метод `recentTacos()`, обрабатывающий HTTP-запросы GET к конечной точке `/api/tacos?recent`, генерирует ответ, содержащий полезную нагрузку в формате JSON с рецептами тако, количество которых не превышает 12. Хорошим началом нам послужит тестовый класс в листинге 12.1.

Листинг 12.1 Использование `WebTestClient` для тестирования `TacoController`

```
package tacos.web.api;

import static org.mockito.ArgumentMatchers.any;
import static org.mockito.Mockito.when;
import java.util.ArrayList;
import java.util.List;
import org.junit.jupiter.api.Test;
import org.mockito.Mockito;
import org.springframework.http.MediaType;
import org.springframework.test.web.reactive.server.WebTestClient;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import tacos.Ingredient;
import tacos.Ingredient.Type;
import tacos.Taco;
import tacos.data.TacoRepository;

public class TacoControllerTest {

    @Test
    public void shouldReturnRecentTacos() {
        Taco[] tacos = {
            testTaco(1L), testTaco(2L),
            testTaco(3L), testTaco(4L), ←———— Создать некоторые тестовые данные
            testTaco(5L), testTaco(6L),
            testTaco(7L), testTaco(8L),
            testTaco(9L), testTaco(10L),
            testTaco(11L), testTaco(12L),
            testTaco(13L), testTaco(14L),
        };
        when(tacosRepository.findAll())
            .thenReturn(Flux.fromIterable(tacos));
        WebTestClient webTestClient = WebTestClient.bindToController(tacoController)
            .build();
        webTestClient.get()
            .uri("/api/tacos?recent")
            .accept(MediaType.APPLICATION_JSON)
            .exchange()
            .expectStatus().isOk()
            .expectBody()
            .jsonPath("$.size()").isEqualTo(12);
    }
}
```

```

        testTaco(15L), testTaco(16L)};
Flux<Taco> tacoFlux = Flux.just(tacos);           Фиктивный экземпляр
TacoRepository tacoRepo = Mockito.mock(TacoRepository.class);   TacoRepository
when(tacoRepo.findAll()).thenReturn(tacoFlux);   ←

WebTestClient testClient = WebTestClient.bindToController(
    new TacoController(tacoRepo))
.build();   ← Создать WebTestClient

testClient.get().uri("/api/tacos?recent")
.exchange()   ← Запросить последние рецепты тако
.expectStatus().isOk()   ←
.expectBody()   Проверить соответствие ответа
    .jsonPath("$.isArray()")
    .jsonPath("$.isNotEmpty()")
    .jsonPath("$.id").isEqualTo(tacos[0].getId().toString())
    .jsonPath("$.name").isEqualTo("Taco 1")
    .jsonPath("$.id").isEqualTo(tacos[1].getId().toString())
    .jsonPath("$.name").isEqualTo("Taco 2")
    .jsonPath("$.id").isEqualTo(tacos[11].getId().toString())
    .jsonPath("$.name").isEqualTo("Taco 12")
    .jsonPath("$.id").doesNotExist();
}

...
}

```

Первым делом метод `shouldReturnRecentTacos()` подготавливает тестовые данные в виде потока `Flux<Taco>`. Затем этот поток `Flux` возвращается из метода `findAll()` фиктивного объекта `TacoRepository`.

Сами объекты `Taco` для публикации в потоке `Flux` создаются с помощью служебного метода `testTaco()`, который получает целое число и создает объект `Taco` с идентификатором и именем, сгенерированными из этого числа. Реализация метода `testTaco()` приводится ниже:

```

private Taco testTaco(Long number) {
    Taco taco = new Taco();
    taco.setId(number != null ? number.toString(): "TESTID");
    taco.setName("Taco " + number);
    List<Ingredient> ingredients = new ArrayList<>();
    ingredients.add(
        new Ingredient("INGA", "Ingredient A", Type.WRAP));
    ingredients.add(
        new Ingredient("INGB", "Ingredient B", Type.PROTEIN));
    taco.setIngredients(ingredients);
    return taco;
}

```

Для простоты все тестовые рецепты тако содержат одни и те же два ингредиента. Но их идентификаторы и названия определяются заданным номером.

Однако вернемся к методу `shouldReturnRecentTacos()`. Здесь мы создали экземпляр `TacoController`, внедрив фиктивный `TacoRepository` в конструктор. Затем контроллер передается в вызов `WebTestClient.bindToController()` для создания экземпляра `WebTestClient`.

После завершения всех настроек можно начинать использовать `WebTestClient` и отправить с его помощью запрос GET к конечной точке `/api/tacos?recent`, после чего проверить соответствия ответа нашим ожиданиям. Вызов `get().uri("/api/tacos?recent")` описывает запрос для обработки. Затем вызов `exchange()` отправляет запрос контроллеру `TacoController`, связанному с `WebTestClient`.

В заключение выполняется проверка соответствия ответа ожиданиям. Вызывая `expectStatus()`, мы убеждаемся, что ответ имеет код состояния HTTP 200 (OK). Далее следует несколько вызовов `jsonPath()`, которые проверяют наличие ожидаемых значений в теле ответа. Последняя проверка убеждается в отсутствии 13-го элемента (с индексом 12 в массиве, где отсчет начинается с нуля), потому что результат никогда не должен содержать больше 12 элементов.

Если возвращаемый ответ в формате JSON имеет сложную структуру с большим количеством данных, то использование `jsonPath()` может быть утомительным. На самом деле я пропустил многие вызовы `jsonPath()` в листинге 12.1 для экономии места. Для случаев, когда использовать `jsonPath()` может быть неудобно, `WebTestClient` предлагает метод `json()`, который принимает строковый параметр с содержимым в формате JSON для сравнения с ответом.

Например, предположим, что мы создали полный ответ в формате JSON в файле с именем `recent-tacos.json` и поместили его в каталог `/tacos` в пути к классам. В этом случае можно переписать проверки `WebTestClient`:

```
ClassPathResource recentsResource =
    new ClassPathResource("/tacos/recent-tacos.json");
String recentsJson = StreamUtils.copyToString(
    recentsResource.getInputStream(), Charset.defaultCharset());

testClient.get().uri("/api/tacos?recent")
    .accept(MediaType.APPLICATION_JSON)
    .exchange()
    .expectStatus().isOk()
    .expectBody()
    .json(recentsJson);
```

Поскольку `json()` принимает строку, мы должны сначала загрузить файл в строку. К счастью, `StreamUtils` в Spring упрощает эту задачу, предлагая функцию `copyToString()`. Стока, возвращаемая функцией `copyToString()`, будет содержать весь текст в формате JSON, который ожидается получить в ответ на запрос. Передача его методу `json()` позволяет убедиться, что контроллер выдает правильный ответ.

Другой вариант, предлагаемый `WebTestClient`: сравнить тело ответа со списком значений. Метод `expectBodyList()` принимает либо

Class, либо ParameterizedTypeReference, определяющий тип элементов в списке, и возвращает объект ListBodySpec, к которому применяются проверки. Используя expectBodyList(), можно переписать тест и сравнить содержимое ответа с теми же исходными данными, на основе которых создавался фиктивный экземпляр TacoRepository:

```
testClient.get().uri("/api/tacos?recent")
    .accept(MediaType.APPLICATION_JSON)
    .exchange()
    .expectStatus().isOk()
    .expectBodyList(Taco.class)
    .contains(Arrays.copyOf(tacos, 12));
```

Здесь тело ответа сравнивается со списком тех же 12 элементов Taco, созданного в начале тестового метода.

12.3.2 Тестирование запросов POST

WebTestClient может тестировать не только запросы GET, но также любые другие запросы HTTP. В табл. 12.1 перечислены методы HTTP и соответствующие им методы WebTestClient.

Таблица 12.1 Методы WebTestClient для тестирования HTTP-запросов, обрабатываемых контроллерами Spring WebFlux

Метод HTTP	Метод WebTestClient
GET	.get()
POST	.post()
PUT	.put()
PATCH	.patch()
DELETE	.delete()
HEAD	.head()

В качестве примера тестирования запросов HTTP другого вида рассмотрим еще один тест для TacoController. На этот раз мы напишем тест для проверки запроса POST к конечной точке создания рецепта тако /api/tacos:

```
@SuppressWarnings("unchecked")
@Test
public void shouldSaveATaco() {
    TacoRepository tacoRepo = Mockito.mock(
        TacoRepository.class); ← Фиктивный экземпляр TacoRepository

    WebTestClient testClient = WebTestClient.bindToController(← Создать WebTestClient
        new TacoController(tacoRepo)).build();

    Mono<Taco> unsavedTacoMono = Mono.just(testTaco(1L));
    Taco savedTaco = testTaco(1L);
    Flux<Taco> savedTacoMono = Flux.just(savedTaco); ← Подготовить тестовые данные
    when(tacoRepo.saveAll(any(Mono.class))).thenReturn(savedTacoMono); ←
```

```

testClient.post() ←
    .uri("/api/tacos")
    .contentType(MediaType.APPLICATION_JSON)
    .body(unsavedTacoMono, Taco.class)
    .exchange()
    .expectStatus().isCreated() ← Проверить ответ
    .expectBody(Taco.class)
    .isEqualTo(savedTaco);
}

```

Так же как в предыдущем примере, метод `shouldSaveATaco()` сначала создает фиктивный экземпляр `TacoRepository`, объект `WebTestClient`, связанный с контроллером, и затем подготавливает некоторые тестовые данные. Потом с помощью `WebTestClient` посыпает POST-запрос конечной точке `/api/tacos` с телом типа `application/json`, содержащим форму создания Тасо, сериализованную в формат JSON, в виде потока `unsavedTacoMono`. После выполнения метода `exchange()` тест проверят статус HTTP 201 (CREATED) в ответе и сравнивает его содержимое с содержимым `savedTacoMono`.

12.3.3 Тестирование с использованием действующего сервера

Тесты, написанные нами до сих пор, основаны на использовании фиктивных объектов, поэтому нам не нужен был действующий сервер. Но контроллеры WebFlux можно также тестировать в контексте настоящего сервера, такого как Netty или Tomcat, и даже с использованием настоящего репозитория и других зависимостей. Проще говоря, мы можем написать интеграционный тест.

Чтобы написать интеграционный тест с `WebTestClient`, начнем с того, что снабдим тестовый класс аннотациями `@RunWith` и `@SpringBootTest`:

```

package tacos;

import java.io.IOException;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.boot.test.context.SpringBootTest.WebEnvironment;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit.jupiter.SpringExtension;
import org.springframework.test.web.reactive.server.WebTestClient;

@ExtendWith(SpringExtension.class)
@SpringBootTest(webEnvironment=WebEnvironment.RANDOM_PORT)
public class TacoControllerWebTest {

    @Autowired
    private WebTestClient testClient;

}

```

Присвоив атрибуту `webEnvironment` значение `WebEnvironment.RANDOM_PORT`, мы требуем от фреймворка Spring запустить действующий сервер, прослушивающий случайно выбранный порт¹.

Обратите внимание, что мы также автоматически связали `WebTestClient` с тестовым классом. Теперь нам не только не нужно создавать его в методах тестирования, но также не нужно указывать полный URL при выполнении запросов, потому что `WebTestClient` будет знать, какой порт использует тестовый сервер. С учетом этих изменений мы можем переписать `shouldReturnRecentTacos()` как интеграционный тест, использующий автоматически созданный `WebTestClient`:

```
@Test
public void shouldReturnRecentTacos() throws IOException {
    testClient.get().uri("/api/tacos?recent")
        .accept(MediaType.APPLICATION_JSON).exchange()
        .expectStatus().isOk()
        .expectBody()
            .jsonPath("$.isArray()")
            .jsonPath("$.length()").isEqualTo(3)
            .jsonPath("$[?(@.name == 'Carnivore')]").exists()
            .jsonPath("$[?(@.name == 'Bovine Bounty')]").exists()
            .jsonPath("$[?(@.name == 'Veg-Out')]").exists();
}
```

Вы наверняка обратили внимание, что в этой новой версии `shouldReturnRecentTacos()` намного меньше кода. Нам больше не нужно создавать `WebTestClient`, потому будет использоваться экземпляр, созданный автоматически. И не нужно создавать фиктивный объект `TacoRepository`, потому что Spring создаст экземпляр `TacoController` и внедрит в него реальный `TacoRepository`. Кроме того, для проверки значений, возвращаемых из базы данных, в этой новой версии метода тестирования используются выражения JSONPath.

`WebTestClient` может пригодиться, если в процессе тестирования понадобится использовать API, предоставляемый контроллером `WebFlux`. Но что делать, если приложение использует какой-то другой API? Давайте обратим внимание на клиентскую сторону реактивного приложения на основе Spring и посмотрим, как `WebClient` работает с реактивными типами, такими как `Mono` и `Flux`.

12.4 Реактивный REST API

В главе 8 мы использовали `RestTemplate` для выполнения клиентских запросов к Taco Cloud API. `RestTemplate` появился еще в версии

¹ Атрибуту `webEnvironment` также можно присвоить значение `WebEnvironment.DEFINED_PORT` и указать номер порта в свойствах атрибута, но обычно поступать так нежелательно, потому что возникает вероятность конфликта портов с уже запущенными серверами.

Spring 3.0. В свое время он использовался для выполнения запросов от имени приложений, которые его используют.

Но все методы `RestTemplate` работают с нереактивными прикладными типами и коллекциями. Это означает, что для работы с данными в ответах на реактивный манер их нужно заключить в потоки `Flux` или `Mono`. И если у вас уже есть поток `Flux` или `Mono` и вам нужно отправить его в запросе `POST` или `PUT`, то перед выполнением запроса вам нужно извлечь данные в нереактивный тип.

Было бы неплохо, если бы имелась возможность использовать `RestTemplate` с реактивными типами. И такая возможность есть! Spring предлагает `WebClient` – реактивную альтернативу `RestTemplate`. `WebClient` позволяет отправлять и получать реактивные типы при работе с внешними API.

Использование `WebClient` сильно отличается от использования `RestTemplate`. Вместо нескольких методов для работы с разными типами запросов `WebClient` предлагает гибкий интерфейс в стиле конструктора, который позволяет описывать и отправлять запросы. Вот как выглядит общий алгоритм работы с `WebClient`:

- создать экземпляр `WebClient` (или внедрить компонент `WebClient`);
- указать HTTP-метод отправляемого запроса;
- указать URI и любые заголовки, которые должны быть в запросе;
- отправить запрос;
- получить и использовать ответ.

Давайте рассмотрим несколько примеров применения `WebClient`, начав с отправки HTTP-запросов `GET`.

12.4.1 Получение ресурсов

В первом примере использования `WebClient` предположим, что нам нужно получить объект `Ingredient` по его идентификатору из Taco Cloud API. При использовании `RestTemplate` можно вызвать метод `getForObject()`. Но при работе с `WebClient` нужно создать запрос, получить ответ, а затем извлечь поток `Mono` с объектом `Ingredient`:

```
Mono<Ingredient> ingredient = WebClient.create()
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .bodyToMono(Ingredient.class);

ingredient.subscribe(i -> { ... });
```

Здесь мы создали новый экземпляр `WebClient` вызовом `create()`. Затем использовали методы `get()` и `uri()`, чтобы определить тип GET запроса и URL `http://localhost:8080/ingredients/{id}`, где на место占位符 `{id}` будет подставлено значение `ingredientId`. Метод `retrieve()` посыпает запрос. Наконец, вызов `bodyToMono()` извлекает содержимое

ответа в поток `Mono<Ingredient>`, к которому можно применить операции, поддерживаемые `Mono`.

Чтобы применить эти дополнительные операции к потоку `Mono`, возвращаемому из `bodyToMono()`, нужно подписаться на него еще до отправки запроса. Работа с запросами, которые могут возвращать коллекции значений, не вызывает никаких сложностей. Например, вот как можно получить все ингредиенты:

```
Flux<Ingredient> ingredients = WebClient.create()
    .get()
    .uri("http://localhost:8080/ingredients")
    .retrieve()
    .bodyToFlux(Ingredient.class);

ingredients.subscribe(i -> { ... });
```

Получение коллекции элементов практически не отличается от получения одного элемента. Разница лишь в том, что для извлечения содержимого ответа вместо `bodyToMono()` используется `bodyToFlux()`, возвращающий поток `Flux`.

Так же как `bodyToMono()`, метод `bodyToFlux()` возвращает поток `Flux`, на который мы еще не подписались. Это позволяет применить к `Flux` дополнительные операции (фильтры, карты и т. д.) до того, как данные начнут проходить через него. Поэтому важно подписаться на полученный поток `Flux`, иначе запрос просто не будет отправлен.

Выполнение запросов с базовым URI

Иногда бывает необходимо выполнить множество различных запросов, имеющих один базовый URI. В таких случаях может быть полезно создать компонент `WebClient` с базовым URI и внедрить его туда, где он необходим. Такой компонент можно объявить следующим образом (в любом классе с аннотацией `@Configuration`):

```
@Bean
public WebClient webClient() {
    return WebClient.create("http://localhost:8080");
}
```

Затем везде, где вам требуется выполнять запросы с этим базовым URI, можно внедрить компонент `WebClient` и использовать его следующим образом:

```
@Autowired
WebClient webClient;

public Mono<Ingredient> getIngredientById(String ingredientId) {
    Mono<Ingredient> ingredient = webClient
        .get()
        .uri("/ingredients/{id}", ingredientId)
        .retrieve()
```

```

    .bodyToMono(Ingredient.class);

    ingredient.subscribe(i -> { ... });
}

```

Так как `WebClient` уже создан, можно сразу приступить к работе, вызвав метод `get()`, а в вызове метода `uri()` указать только путь относительно базового URI.

Выполнение длительных запросов

При работе с сетями важно помнить, что они не всегда надежны и не так быстры, как хотелось бы. Кроме того, удаленный сервер может слишком медленно обрабатывать запросы. В идеале ответ на запрос к удаленной службе должен возвращаться за разумное время. Но в случае каких-либо проблем хотелось бы, чтобы клиент не «зависал» в ожидании ответа.

Дабы избежать зависания клиентов из-за медленной сети или службы, можно использовать метод `timeout()` потоков `Flux` и `Mono`, чтобы ограничить время ожидания публикации данных. Для примера давайте посмотрим, как можно использовать `timeout()` при получении данных об ингредиентах:

```

Flux<Ingredient> ingredients = webclient
    .get()
    .uri("/ingredients")
    .retrieve()
    .bodyToFlux(Ingredient.class);

ingredients
    .timeout(Duration.ofSeconds(1))
    .subscribe(
        i -> { ... },
        e -> {
            // обработка ошибки тайм-аута
        });
}

```

Как видите, перед подпиской на `Flux` здесь вызывается метод `timeout()`, которому задается максимальное время ожидания, равное 1 с. Если запрос будет выполнен менее чем за 1 секунду, то никаких проблем не возникнет. Но если ответ на запрос задержится больше, чем на 1 секунду, то заданный тайм-аут истечет и будет вызван обработчик ошибок, заданный вторым параметром в вызове `subscribe()`.

12.4.2 Отправка ресурсов

Отправка данных с помощью `WebClient` мало отличается от получения. Для примера предположим, что у нас есть `Mono<Ingredient>` и мы хотим отправить запрос `POST` с объектом `Ingredient`, опубликованным в потоке `Mono`, к конечной точке `/ingredients`. Для этого нужно всего

лишь вызвать метод `post()` вместо `get()` и указать, что для заполнения тела запроса должен использоваться поток `Mono`, вызвав `body()`:

```
Mono<Ingredient> ingredientMono = Mono.just(
    new Ingredient("INGC", "Ingredient C", Ingredient.Type.VEGGIES));
Mono<Ingredient> result = webClient
    .post()
    .uri("/ingredients")
    .body(ingredientMono, Ingredient.class)
    .retrieve()
    .bodyToMono(Ingredient.class);

result.subscribe(i -> { ... });
```

Если вместо потока `Mono` или `Flux` имеется только прикладной объект, то можно использовать метод `bodyValue()`. Например, предположим, что вместо `Mono<Ingredient>` у нас есть объект `Ingredient`, который нужно отправить в теле запроса. Вот как это можно сделать:

```
Ingredient ingredient = ...;

Mono<Ingredient> result = webClient
    .post()
    .uri("/ingredients")
    .bodyValue(ingredient)
    .retrieve()
    .bodyToMono(Ingredient.class);

result.subscribe(i -> { ... });
```

Чтобы обновить существующий ресурс `Ingredient` с помощью запроса PUT, нужно вместо метода `post()` вызвать `put()` и задать соответствующий путь в URI:

```
Mono<Void> result = webClient
    .put()
    .uri("/ingredients/{id}", ingredient.getId())
    .bodyValue(ingredient)
    .retrieve()
    .bodyToMono(Void.class);

result.subscribe();
```

Ответы на запросы PUT обычно не несут полезных данных, поэтому в таких случаях следует попросить `bodyToMono()` вернуть Моно типа `Void`. Запрос будет отправлен сразу после подписки на этот Моно.

12.4.3 Удаление ресурсов

`WebClient` также позволяет удалять ресурсы, предоставляя метод `delete()`. Например, следующий код удалит ингредиент с указанным идентификатором:

```
Mono<Void> result = webClient
    .delete()
    .uri("/ingredients/{id}", ingredientId)
    .retrieve()
    .bodyToMono(Void.class);

result.subscribe();
```

Ответы на запросы DELETE, как и на запросы PUT, обычно не несут полезных данных. Поэтому для отправки запроса снова нужно попросить bodyToMono() вернуть Mono типа Void.

12.4.4 Обработка ошибок

Все примеры использования WebClient, приводившиеся до сих пор, имели счастливый конец; мы не получали ответов с кодами состояния из диапазона 400 или 500. Если сервер вернет код ошибки любого типа, то WebClient зафиксирует сбой и продолжит работу, как ни в чем не бывало.

Чтобы обработать такие ошибки, можно вызвать метод `onStatus()` и указать, как следует обрабатывать различные коды состояния HTTP. Метод `onStatus()` принимает две функции: функцию-предикат для проверки статуса HTTP и функцию, возвращающую `Mono<Throwable>`.

Для демонстрации использования метода `onStatus()` с целью создания своего обработчика ошибок рассмотрим следующее определение `WebClient`, который получает ингредиент по его идентификатору:

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .bodyToMono(Ingredient.class);
```

Если значение `inventoryId` соответствует существующему ресурсу ингредиента, то в результирующем потоке `Mono` будет возвращаться объект `Ingredient`. Но что случится, если в базе данных не окажется искомого ингредиента?

Если подписка на поток `Mono` или `Flux` может закончиться ошибкой, важно зарегистрировать в вызове `subscribe()` получателей ошибок и данных, как показано ниже:

```
ingredientMono.subscribe(
    ingredient -> {
        // обработать полученный ингредиент
        ...
    },
    e -> {
        // обработать ошибку
        ...
});
```

Если искомый ресурс ингредиента существует, то будет вызвано первое лямбда-выражение (получатель данных), которое получит соответствующий объект `Ingredient`. Если искомый ресурс отсутствует, то на запрос вернется ответ кодом состояния HTTP 404 (NOT FOUND) и будет вызвано второе лямбда-выражение (получатель ошибок), которое получит исключение по умолчанию `WebClientResponseException`.

Самая большая проблема с `WebClientResponseException` в том, что это довольно обобщенное исключение, мало говорящее о фактической причине ошибки. Его имя говорит лишь о том, что на запрос, выполненный с помощью `WebClient`, вернулся ответ с признаком ошибки, и вам придется покопаться в недрах `WebClientResponseException`, чтобы узнать причину. В любом случае было бы неплохо, если бы исключение, передаваемое получателю ошибок, было более специфичным.

Добавляя свой обработчик ошибок, можно определить код, который преобразует код состояния в экземпляр `Throwable` по вашему выбору. Допустим, нам нужно, чтобы неудачная попытка получить ресурс ингредиента приводила к получению `Mono` с ошибкой `UnknownIngredientException`. Чтобы добиться этого, можно добавить следующий вызов `onStatus()` после вызова `retrieve()`:

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .onStatus(HttpStatus::is4xxClientError,
        response -> Mono.just(new UnknownIngredientException()))
    .bodyToMono(Ingredient.class);
```

Первый аргумент в вызове `onStatus()` – это предикат, который получает `HttpStatus` и возвращает `true`, если код состояния подлежит обработке. Если получен код состояния, подлежащий обработке, то ответ будет передан для обработки функции, заданной во втором аргументе, в конечном итоге возвращающей `Mono` типа `Throwable`.

В этом примере, если код состояния попадает в диапазон кодов 4XX (например, ошибка клиента), будет возвращен поток `Mono` с исключением `UnknownIngredientException`. Это приводит к тому, что `ingredientMono` выдаст это исключение.

Обратите внимание, что `HttpStatus::is4xxClientError` – это ссылка на метод `is4xxClientError` класса `HttpStatus`. Именно этот метод будет вызываться для проверки данного объекта `HttpStatus`. При желании можно использовать другой метод `HttpStatus` или передать свою функцию в виде лямбда-выражения или ссылки на метод, которая возвращает логическое значение.

Например, можно еще точнее обрабатывать ошибки, специально проверяя состояние HTTP 404 (NOT FOUND), изменив вызов `onStatus()`, как показано ниже:

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .onStatus(status -> status == HttpStatus.NOT_FOUND,
              response -> Mono.just(new UnknownIngredientException()))
    .bodyToMono(Ingredient.class);
```

Также стоит отметить, что можно добавить столько вызовов `onStatus()`, сколько нужно для обработки любых кодов состояния HTTP, которые могут вернуться в ответе.

12.4.5 Обмен запросами

До сих пор для обозначения отправки запроса при работе с `WebClient` мы использовали метод `retrieve()`. Метод `retrieve()` возвращает экземпляр `ResponseSpec`, с помощью которого можно обрабатывать ответ, вызывая такие методы, как `onStatus()`, `bodyToFlux()` и `bodyToMono()`. Объект `ResponseSpec` хорошо подходит в простых случаях, но он имеет некоторые ограничения. Например, если понадобится проанализировать заголовки ответа или значения cookie, то `ResponseSpec` вам не подойдет.

В случаях, когда `ResponseSpec` не подходит, можно попробовать вместо `retrieve()` вызвать `exchangeToMono()` или `exchangeToFlux()`. Метод `exchangeToMono()` возвращает поток `Mono` типа `ClientResponse`, к которому можно применять реактивные операции для проверки и использования данных из ответа, включая полезную нагрузку, заголовки и cookie. Метод `exchangeToFlux()` работает почти так же, но возвращает поток `Flux` типа `ClientResponse`, что позволяет работать с несколькими элементами данных в ответе.

Прежде чем перейти к обзору отличий `exchangeToMono()` и `exchangeToFlux()` от `retrieve()`, посмотрим сначала их сходства. Следующий фрагмент кода использует `WebClient` и `exchangeToMono()` для получения одного ингредиента по идентификатору:

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .exchangeToMono(cr -> cr.bodyToMono(Ingredient.class));
```

Он примерно эквивалентен следующему фрагменту, в котором используется функция `retrieve()`:

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .retrieve()
    .bodyToMono(Ingredient.class);
```

Во втором фрагменте мы вызываем метод `bodyToMono()` объекта `ResponseSpec`, чтобы получить `Mono<Ingredient>`, а в первом – метод

`exchangeToMono()`, чтобы получить поток `Mono<ClientResponse>`, к которому можно применить функцию `flatMap()` и преобразовать `ClientResponse` в тот же самый поток `Mono<Ingredient>`.

Теперь посмотрим, чем `exchangeToMono()` отличается от `retrieve()`. Предположим, что ответ на запрос может включать заголовок `X_UNAVAILABLE` со значением `true`, указывающим, что (по какой-то причине) искомый ингредиент недоступен. Предположим также, что если этот заголовок существует, то результирующий поток `Mono` должен быть пустым и ничего не возвращать. Реализовать это можно, добавив еще один вызов `flatMap()`, но сделать это с `WebClient` намного проще:

```
Mono<Ingredient> ingredientMono = webClient
    .get()
    .uri("http://localhost:8080/ingredients/{id}", ingredientId)
    .exchangeToMono(cr -> {
        if (cr.headers().header("X_UNAVAILABLE").contains("true")) {
            return Mono.empty();
        }
        return Mono.just(cr);
    })
    .flatMap(cr -> cr.bodyToMono(Ingredient.class));
```

Новый вызов `flatMap()` проверяет заголовки объекта `ClientRequest`, пытаясь найти заголовок с именем `X_UNAVAILABLE` и значением `true`. Если такой заголовок будет найден, то возвращается пустой поток `Mono`. Иначе возвращается новый поток `Mono`, содержащий `ClientResponse`. В любом случае возвращаемый поток `Mono` преобразуется в другой поток `Mono`, с которым будет работать следующий вызов `flatMap()`.

12.5 Защита реактивного веб-API

С тех пор, как появился фреймворк Spring Security (и даже немногим раньше, когда он был известен под названием Acegi Security), его модель безопасности основывалась на фильтрах сервлетов. И не безосновательно. Если нужно перехватить запрос с использованием веб-фреймворка на основе сервлетов, чтобы убедиться, что отправитель имеет надлежащие полномочия, то фильтр сервлета является очевидным выбором. Но Spring WebFlux внес свою изюминку в этот подход.

Веб-приложения, реализованные с использованием Spring WebFlux, могут работать без сервлетов. На самом деле реактивное веб-приложение с большей вероятностью будет построено на основе Netty или каком-нибудь другом сервере, не являющемся сервлем. Означает ли это, что Spring Security, основанный на фильтрах сервлетов, нельзя использовать для защиты приложений Spring WebFlux?

Использовать фильтры сервлетов для защиты приложений Spring WebFlux действительно нельзя. Но Spring Security по-прежнему решает стоящие перед ним задачи. Начиная с версии 5.0.0 появилась возможность использовать Spring Security для защиты не только приложений Spring MVC на основе сервлетов, но также реактивных приложений Spring WebFlux. Защита реактивных приложений основывается на Spring WebFilter, аналоге фильтров сервлетов, не зависящем от наличия поддержки сервлетов.

Еще более примечательно, что модель конфигурации реактивной реализации Spring Security не сильно отличается от той, что мы видели в главе 4. Фактически, в отличие от Spring WebFlux, имеющего отдельную зависимость от Spring MVC, Spring Security подключается к приложению с использованием все той же начальной зависимости Spring Boot, независимо от того, строится это приложение на основе Spring MVC или Spring WebFlux. Напомню, как выглядит эта начальная зависимость:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Однако реактивная и нереактивная модели конфигурации Spring Security имеют небольшие различия, поэтому я предлагаю бегло просмотреть эти различия, сравнив эти две модели.

12.5.1 Реактивная модель настройки безопасности

Напомню, что для настройки защиты с использованием Spring Security веб-приложения на основе Spring MVC обычно определяют класс конфигурации, наследующий `WebSecurityConfigurerAdapter` и снабженный аннотацией `@EnableWebSecurity`. Класс конфигурации должен переопределять метод `configuration()`, в котором настраиваются параметры безопасности, например перечисляются конечные точки, требующие авторизации. Следующий простой класс конфигурации для настройки Spring Security поможет вам вспомнить, как настраивается безопасность в нереактивном приложении Spring MVC:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
            .antMatchers("/api/tacos", "/orders").hasAuthority("USER")
            .antMatchers("/**").permitAll();
    }
}
```

Теперь посмотрим, как выглядит аналогичная конфигурация для реактивного приложения Spring WebFlux. В листинге 12.2 показан реактивный класс конфигурации безопасности, примерно эквивалентный классу конфигурации из предыдущего примера.

Листинг 12.2 Конфигурация Spring Security для приложения Spring WebFlux

```
@Configuration
@EnableWebFluxSecurity
public class SecurityConfig {

    @Bean
    public SecurityWebFilterChain securityWebFilterChain(
            ServerHttpSecurity http) {
        return http
            .authorizeExchange()
            .pathMatchers("/api/tacos", "/orders").hasAuthority("USER")
            .anyExchange().permitAll()
            .and()
            .build();
    }

}
```

Как видите, эти классы имеют много общего, но есть и немало отличий. Вместо `@EnableWebSecurity` этот новый класс конфигурации отмечен аннотацией `@EnableWebFluxSecurity`. Кроме того, класс в листинге 12.2 не наследует `WebSecurityConfigurerAdapter` или какой-то другой базовый класс, поэтому он не переопределяет метод `configure()`.

Вместо метода `configure()` здесь объявляется метод `securityWebFilterChain()`, конструирующий bean-компонент типа `SecurityWebFilterChain`. Тело метода `securityWebFilterChain()` похоже на метод `configure()` в предыдущей конфигурации, но есть некоторые тонкие отличия.

Прежде всего конфигурация объявляется с использованием объекта `ServerHttpSecurity` вместо `HttpSecurity`. Используя `ServerHttpSecurity`, можно вызвать метод `authorizeExchange()`, примерно эквивалентный методу `authorizeRequests()`, и определить параметры безопасности на уровне запроса.

ПРИМЕЧАНИЕ `ServerHttpSecurity` появился в версии Spring Security 5 и является реактивным аналогом `HttpSecurity`.

Для сопоставления путей по-прежнему можно использовать подстановочные знаки в стиле Ant, но делать это нужно с помощью метода `pathMatchers()` вместо `antMatchers()`. Также появилась возможность не указывать универсальный путь `/**` в стиле Ant, потому что его возвращает `anyExchange()`.

Наконец, поскольку `SecurityWebFilterChain` объявляется как компонент, необходимо вызвать метод `build()`, чтобы собрать все правила безопасности в возвращаемый объект `SecurityWebFilterChain`. Кроме этих небольших отличий, настройка безопасности веб-приложений Spring WebFlux очень похожа на настройку безопасности для Spring MVC. А как реализуется поддержка учетных записей пользователей?

12.5.2 Настройка реактивной службы учетных записей

При наследовании `WebSecurityConfigurerAdapter` достаточно переопределить один метод `configure()`, чтобы объявить в нем правила веб-безопасности, и другой метод `configure()`, чтобы настроить логику аутентификации, обычно путем определения объекта `UserDetails`. Дабы вспомнить, как это выглядит, взгляните на следующий переопределенный метод `configure()`, который использует объект `UserRepository`, внедренный в анонимную реализацию `UserDetailsService`, осуществляющую поиск пользователя по его имени:

```
@Autowired
UserRepository userRepo;

@Override
protected void
configure(AuthenticationManagerBuilder auth)
    throws Exception {
    auth
        .userDetailsService(new UserDetailsService() {
            @Override
            public UserDetails loadUserByUsername(String username)
                throws UsernameNotFoundException {
                User user = userRepo.findByUsername(username)
                if (user == null) {
                    throw new UsernameNotFoundException(
                        username + " not found")
                }
                return user.toUserDetails();
            }
        });
}
```

В этой нереактивной конфигурации мы переопределили единственный метод, необходимый службе `UserDetailsService`: `loadUserByUsername()`. Внутри метода мы используем внедренный `UserRepository` для поиска пользователя по его имени. Если пользователь не найден, то генерируется исключение `UsernameNotFoundException`. Если найден – вызывается вспомогательный метод `toUserDetails()`, возвращающий соответствующий объект `UserDetails`.

В реактивной конфигурации мы не переопределяем метод `configure()`, а просто объявляем компонент `ReactiveUserDetailsService`

vice – реактивный эквивалент UserDetailsService. Так же как UserDetailsService, компонент ReactiveUserDetailsService требует реализации только одного метода – findByUsername(), который возвращает Mono<UserDetails> вместо простого объекта UserDetails.

В следующем примере компонент ReactiveUserDetailsService использует внедренный UserRepository, который, как предполагается, является реактивным репозиторием Spring Data (о реактивных репозиториях мы поговорим в следующей главе):

```
@Bean
public ReactiveUserDetailsService userDetailsService(
    UserRepository userRepo) {
    return new ReactiveUserDetailsService() {
        @Override
        public Mono<UserDetails> findByUsername(String username) {
            return userRepo.findByUsername(username)
                .map(user -> {
                    return user.toUserDetails();
                });
        }
    };
}
```

Наш метод findByUsername() возвращает Mono<UserDetails>, но метод UserRepository.findByUsername() возвращает Mono<User>. Так как это поток Mono, к нему можно применить различные операции, и в данном случае применяется операция map(), отображающая Mono<User> в Mono<UserDetails>.

В этом случае в операцию map() передается лямбда-выражение, вызывающее вспомогательный метод toUserDetails() для преобразования объекта User, опубликованного в потоке Mono, в объект UserDetails. Как следствие операция map() возвращает Mono<UserDetails>, что и требуется от переопределенного метода ReactiveUserDetailsService.findByUsername(). Если findByUsername() не найдет соответствующего пользователя в репозитории, то вернет пустой поток Mono, чтобы показать, что попытка аутентификации потерпела неудачу.

Итоги

- Spring WebFlux – это реактивный веб-фреймворк, предлагающий модель программирования, подобную модели Spring MVC, и даже использующий множество похожих аннотаций.
- Spring также поддерживает модель функционального программирования как альтернативу модели программирования на основе аннотаций.

- Реактивные контроллеры можно тестировать с помощью `WebTestClient`.
- На стороне клиента Spring предлагает `WebClient` – реактивный аналог `RestTemplate`.
- Фреймворк `WebFlux` оказывает существенное влияние на базовые механизмы защиты веб-приложений, однако реактивная модель программирования, реализованная в `Spring Security 5`, не сильно отличается от нереактивной модели `Spring MVC`.

17 Реактивное хранение данных

В этой главе рассматриваются следующие темы:

- организация реактивного реляционного хранилища с R2DBC;
- определение реактивных репозиториев на основе MongoDB и Cassandra;
- тестирование реактивных репозиториев.

Научная фантастика научила нас: если мы хотим что-то улучшить в прошлом, то для этого нужно просто совершить небольшое путешествие во времени. Этот прием сработал в «Назад в будущее», в нескольких эпизодах сериалов «Звездный путь», «Мстители: финал» и «22.11.63» Стивена Кинга. (Ладно, ладно. В последнем случае получилось не особенно хорошо, но сама идея, думаю, вам понятна.)

В этой главе мы вернемся к главам 3 и 4 и еще раз обсудим вопросы использования реляционных баз данных, а также MongoDB и Cassandra. Но на этот раз мы внесем некоторые усовершенствования, добавив поддержку реактивности из Spring Data, что позволит нам работать с этими репозиториями неблокирующим образом.

Начнем с использования Spring Data R2DBC – реактивной альтернативы Spring Data JDBC для взаимодействий с реляционными базами данных.

13.1 R2DBC

Reactive Relational Database Connectivity, или R2DBC (<https://r2dbc.io/>), – относительно новый инструмент для работы с реляционными базами данных с применением реактивных типов. По сути, это реактивная альтернатива JDBC, обеспечивающая возможность взаимодействий без блокировки с обычными реляционными базами данных, такими как MySQL, PostgreSQL, H2 и Oracle. Поскольку этот механизм основан на библиотеке Reactive Streams, он сильно отличается от JDBC и фактически является отдельной спецификацией, не связанной с Java SE.

Spring Data R2DBC – это подпроект Spring Data, предлагающий автоматическую поддержку репозиториев, почти так же, как Spring Data JDBC, с которым мы познакомились в главе 3. Однако, в отличие от Spring Data JDBC, Spring Data R2DBC не требует строгого соблюдения концепции предметного проектирования. На самом деле, как вы вскоре увидите, попытка сохранить данные через агрегатный корень в Spring Data R2DBC требует лишь немногим больше усилий, чем в Spring Data JDBC.

Чтобы получить возможность пользоваться ресурсами Spring Data R2DBC, нужно добавить начальную зависимость в спецификацию сборки проекта. Для проекта Maven зависимость выглядит так:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-r2dbc</artifactId>
</dependency>
```

Также можно воспользоваться флагком **Spring Data R2DBC** в веб-приложении Initializr.

Помимо этого, понадобится реляционная база данных для хранения данных и соответствующий драйвер R2DBC. В нашем проекте мы будем использовать базу данных H2, хранящуюся в оперативной памяти. Поэтому добавим еще две зависимости: саму библиотеку базы данных H2 и драйвер H2 R2DBC:

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.r2dbc</groupId>
    <artifactId>r2dbc-h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Если вы используете другую базу данных, то добавьте зависимость с драйвером R2DBC для выбранной вами базы данных.

Теперь, добавив необходимые зависимости, посмотрим, как работает Spring Data R2DBC. Начнем с определения объектов предметной области.

13.1.1 Определение сущностей предметной области для R2DBC

С целью знакомства со Spring Data R2DBC мы повторно реализуем уровень хранения данных в приложении Taco Cloud и сосредоточимся исключительно на компонентах, необходимых для хранения рецептов тако и заказов. Для этого мы определим сущности предметной области, соответствующие объектам TacoOrder, Taco и Ingredient, а также репозитории для каждой из них.

Для начала определим класс сущности предметной области Ingredient (листинг 13.1).

Листинг 13.1 Класс сущности Ingredient для сохранения с использованием R2DBC

```
package tacos;

import org.springframework.data.annotation.Id;
import lombok.Data;
import lombok.EqualsAndHashCode;
import lombok.NoArgsConstructor;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

@Data
@NoArgsConstructor
@RequiredArgsConstructor
@EqualsAndHashCode(exclude = "id")
public class Ingredient {

    @Id
    private Long id;

    private @NonNull String slug;

    private @NonNull String name;
    private @NonNull Type type;

    public enum Type {
        WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
    }
}
```

Как видите, эта реализация не сильно отличается от других воплощений класса Ingredient, созданных нами ранее. Тем не менее обратите внимание на следующие два важных отличия:

- Spring Data R2DBC требует, чтобы свойства имели методы доступа, поэтому большинство свойств определено как нефинальные. Но чтобы помочь библиотеке Lombok создать обязательный конструктор с аргументами, мы снабдили большинство свойств аннотацией `@NonNull`. Благодаря этому Lombok и аннотация `@RequiredArgsConstructor` включат эти свойства в конструктор;
- при попытке сохранить в репозиторий Spring Data R2DBC объект с ненулевым значением свойства `id` операция будет рассматриваться как изменение имеющейся сущности. Свойство `id` класса `Ingredient` раньше было объявлено с типом `String`. Но при использовании Spring Data R2DBC это приведет к ошибке. Поэтому здесь мы преобразовали свойство идентификатора в новое строковое свойство с именем `slug`, которое является псевдоидентификатором ингредиента, а для идентификации в базе данных используем целочисленное свойство `Long id`, значение которого генерируется самой базой данных.

Соответствующая таблица в базе данных определена в `schema.sql`:

```
create table Ingredient (
    id identity,
    slug varchar(4) not null,
    name varchar(25) not null,
    type varchar(10) not null
);
```

Класс сущности `Taco` тоже очень похож на свой аналог Spring Data JDBC, как показано в листинге 13.2.

Листинг 13.2 Класс сущности Taco для сохранения с использованием R2DBC

```
package tacos;

import java.util.HashSet;
import java.util.Set;
import org.springframework.data.annotation.Id;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.NonNull;
import lombok.RequiredArgsConstructor;

@Data
@NoArgsConstructor
@RequiredArgsConstructor
public class Taco {

    @Id
    private Long id;

    private @NonNull String name;
```

```

private Set<Long> ingredientIds = new HashSet<>();

public void addIngredient(Ingredient ingredient) {
    ingredientIds.add(ingredient.getId());
}

}

```

Так же как в случае с классом `Ingredient`, мы должны обеспечить создание методов доступа к свойствам объекта, именно поэтому здесь применяется аннотация `@NotNull` вместо спецификатора `final`.

Обратите особое внимание, что вместо коллекции объектов `Ingredient` в классе `Taco` используется множество `Set<Long>`, ссылающееся на идентификаторы объектов `Ingredient`. Множество `Set` было выбрано вместо списка `List`, чтобы гарантировать уникальность. Но почему мы использовали `Set<Long>`, а не `Set<Ingredient>?`

В отличие от других проектов Spring Data, Spring Data R2DBC в настоящее время не поддерживает прямые отношения между сущностями (по крайней мере пока). В относительно новом проекте Spring Data R2DBC все еще не решены некоторые проблемы обработки отношений неблокирующим способом. Но ситуация может измениться в будущих версиях Spring Data R2DBC.

А до тех пор мы не можем использовать в `Taco` ссылки на ингредиенты и ожидать, что механизм хранения обработает их правильно. Поэтому нам остается только несколько вариантов представления отношений:

- определить сущности со ссылками на идентификаторы связанных объектов. В этом случае соответствующий столбец в таблице базы данных должен быть определен как массив, если это возможно. Столбцы-массивы поддерживаются базами данных H2 и PostgreSQL, но не поддерживаются многими другими. Кроме того, даже если база данных поддерживает столбцы-массивы, она может не поддерживать определение внешних ключей на основе этих столбцов, ссылающихся на другие таблицы, что делает невозможным обеспечение ссылочной целостности;
- определить сущности и соответствующие им таблицы так, чтобы они идеально соответствовали друг другу. Для коллекций это означает, что сущность, на которую указывает ссылка, должна иметь столбец с обратной ссылкой на ссылающуюся таблицу. Например, в таблице для сущностей `Taco` должен быть столбец, ссылающийся на сущность `TacoOrder`, частью которой является `Taco`;
- сериализовать сущности в формат в JSON и хранить определение JSON в столбце VARCHAR. Этот прием особенно хорош, если нет необходимости запрашивать объекты по ссылкам. Однако у него есть потенциальное ограничение на размер сериализованных объектов JSON, обусловленное ограничением длины соответствующего столбца VARCHAR. Более того, в этом случае нет никакой возможности использовать схему базы данных, чтобы

гарантирует ссылочную целостность, потому что объекты, на которые указывают ссылки, будут храниться как простое строковое значение (которое может содержать что угодно).

Ни один из этих вариантов не идеален, но, взвесив все «за» и «против», я решил выбрать первый вариант. Класс Taco имеет свойство `Set<Long>`, ссылающееся на один или несколько идентификаторов ингредиентов. Это означает, что соответствующая таблица должна иметь столбец-массив для хранения этих идентификаторов. В базе данных H2 таблица Taco определяется следующим образом:

```
create table Taco (
    id identity,
    name varchar(50) not null,
    ingredient_ids array
);
```

Тип `array` столбца `ingredient_ids` специфичен для H2. Для PostgreSQL этот столбец можно определить с типом `integer[]`. Чтобы узнать, как определить столбцы-массивы в выбранной вами базе данных, обращайтесь к документации по ней. Но не забывайте, что не все базы данных поддерживают столбцы-массивы, поэтому для моделирования отношений вам может потребоваться выбрать другой вариант из перечисленных выше.

Наконец, класс `TacoOrder`, как показано в листинге 13.3, определяется с учетом многих особенностей, которые мы уже использовали при определении сущностей предметной области Taco и Ingredient.

Листинг 13.3 Класс сущности TacoOrder для сохранения с использованием R2DBC

```
package tacos;

import java.util.LinkedHashSet;
import java.util.Set;
import org.springframework.data.annotation.Id;
import lombok.Data;

@Data
public class TacoOrder {

    @Id
    private Long id;

    private String deliveryName;
    private String deliveryStreet;
    private String deliveryCity;
    private String deliveryState;
    private String deliveryZip;
    private String ccNumber;
    private String ccExpiration;
    private String ccCVV;
```

```

private Set<Long> tacoIds = new LinkedHashSet<>();

private List<Taco> tacos = new ArrayList<>();
public void addTaco(Taco taco) {
    this.tacos.add(taco);
}
}

```

Как видите, помимо чуть большего количества свойств, класс TacoOrder следует той же схеме, что и класс Taco. Он ссылается на дочерние объекты Taco через свойство Set<Long>. Однако чуть ниже вы увидите, как получить объекты Taco, опираясь на TacoOrder, несмотря на то что Spring Data R2DBC не поддерживает прямых отношений. Вот как выглядит схема базы данных для таблицы Taco_Order:

```

create table Taco_Order (
    id identity,
    delivery_name varchar(50) not null,
    delivery_street varchar(50) not null,
    delivery_city varchar(50) not null,
    delivery_state varchar(2) not null,
    delivery_zip varchar(10) not null,
    cc_number varchar(16) not null,
    cc_expiration varchar(5) not null,
    cc_cvv varchar(3) not null,
    taco_ids array
);

```

Подобно таблице Taco, которая ссылается на ингредиенты через столбец-массив, таблица TacoOrder ссылается на дочерние рецепты Taco через столбец `taco_ids`, также объявленный как столбец-массив. И снова эта схема предназначена исключительно для базы данных H2; за подробной информацией о поддержке и создании столбцов-массивов в другой базе данных обращайтесь к документации по ней.

Часто промышленные приложения определяют схему другими средствами, и определения схем, как в сценарии выше, годятся разве что для тестирования. По этой причине данный компонент определен в конфигурации, которая загружается только при выполнении автоматизированных тестов и недоступен в контексте времени выполнения приложения. Мы рассмотрим пример тестирования репозиториев R2DBC, когда определим все необходимые службы.

Отметьте также, что этот компонент использует только файл `schemas.sql` из корня пути к классам (в каталоге проекта `src/main/resources`). Если для инициализации данных потребуется использовать другие сценарии SQL, добавьте дополнительные объекты ResourceDatabasePopulator вызовом `populator.addPopulators()`.

Теперь, определив наши сущности и соответствующие им схемы, создадим репозитории, куда будут сохраняться и откуда извлекаться данные.

13.1.2 Определение реактивных репозиториев

В главах 3 и 4 мы определили репозитории как интерфейсы, наследующие интерфейс `CrudRepository` из Spring Data. Но этот базовый интерфейс имел дело с отдельными объектами и коллекциями `Iterable`. Реактивный репозиторий, напротив, будет иметь дело с объектами `Mono` и `Flux`.

По этой причине для определения реактивных репозиториев Spring Data предлагает интерфейс `ReactiveCrudRepository`. `ReactiveCrudRepository` очень похож на `CrudRepository`. Чтобы создать репозиторий, нужно определить интерфейс, наследующий `ReactiveCrudRepository`, например:

```
package tacos.data;

import org.springframework.data.repository.reactive.ReactiveCrudRepository;
import tacos.TacoOrder;

public interface OrderRepository
    extends ReactiveCrudRepository<TacoOrder, Long> {
}
```

Единственное отличие, видимое на первый взгляд, между интерфейсом `OrderRepository` и интерфейсами, которые мы определили в главах 3 и 4, заключается в наследовании `ReactiveCrudRepository` вместо `CrudRepository`. Но важно также отметить, что его методы возвращают типы `Mono` и `Flux` вместо одиночного объекта `TacoOrder` или `Iterable<TacoOrder>`. Примерами могут служить методы `findById()`, возвращающий `Mono<TacoOrder>`, и `findAll()`, возвращающий `Flux<TacoOrder>`.

Чтобы увидеть этот реактивный репозиторий в действии, предположим, что нам нужно получить все объекты `TacoOrder` и вывести имена получателей заказов в стандартный вывод. Сделать это можно, как показано в листинге 13.4.

Листинг 13.4 Вызов метода реактивного репозитория

```
@Autowired
OrderRepository orderRepo;

...

orderRepository.findAll()
    .doOnNext(order -> {
        System.out.println(
            "Deliver to: " + order.getDeliveryName());
    })
    .subscribe();
```

Здесь вызов `findAll()` возвращает поток `Flux<TacoOrder>`, к которому мы применили операцию `doOnNext()` для вывода имени получа-

теля заказа. Наконец, вызов `subscribe()` запускает передачу данных через Flux.

В примере использования Spring Data JDBC в главе 3 объект `TacoOrder` был корнем агрегата, а `Taco` – одной из составляющих агрегата. Поэтому объекты `Taco` сохранялись как часть `TacoOrder`, и не было необходимости определять отдельный репозиторий для хранения `Taco`. Но Spring Data R2DBC не поддерживает корни агрегатов, поэтому нам необходимо создать `TacoRepository` для хранения объектов `Taco`, определение которого приводится в листинге 13.5.

Листинг 13.5 Реактивный репозиторий для хранения объектов Taco

```
package tacos.data;

import org.springframework.data.repository.reactive.ReactiveCrudRepository;
import tacos.Taco;

public interface TacoRepository
    extends ReactiveCrudRepository<Taco, Long> {
}
```

Как видите, интерфейс `TacoRepository` мало чем отличается от `OrderRepository`. Он наследует `ReactiveCrudRepository`, предоставляющий реактивные типы для работы с хранилищем объектов `Taco`. Здесь нет ничего нового.

Интерфейс `IngredientRepository`, напротив, более интересный (листинг 13.6).

Листинг 13.6 Реактивный репозиторий для хранения объектов Ingredient

```
package tacos.data;

import org.springframework.data.repository.reactive.ReactiveCrudRepository;
import reactor.core.publisher.Mono;
import tacos.Ingredient;

public interface IngredientRepository
    extends ReactiveCrudRepository<Ingredient, Long> {
    Mono<Ingredient> findBySlug(String slug);
}
```

Подобно двум другим нашим реактивным репозиториям, `IngredientRepository` наследует `ReactiveCrudRepository`. Но, как вы помните, нам может понадобиться возможность искать объекты `Ingredient` по значению свойства `slug`, поэтому `IngredientRepository` включает метод `findBySlug()`, возвращающий поток `Mono<Ingredient>`¹.

¹ В репозитории на основе JDBC (глава 3) в этом методе не было необходимости, потому что роль идентификатора и краткого названия у нас играло поле `id`.

Теперь посмотрим, как написать тесты для проверки работоспособности наших репозиториев.

13.1.3 Тестирование репозиториев R2DBC

Spring Data R2DBC включает поддержку интеграционного тестирования реактивных репозиториев. В частности, аннотация `@DataR2dbcTest` заставляет Spring создать контекст приложения со сгенерированными репозиториями Spring Data R2DBC в виде компонентов, которые можно внедрить в тестовый класс. Наряду с объектом `StepVerifier`, который мы использовали в предыдущих главах, эта поддержка позволяет писать автоматизированные тесты для всех созданных репозиториев.

Для краткости мы рассмотрим только один тестовый класс: `IngredientRepositoryTest`. Он проверит способность репозитория `IngredientRepository` сохранять объекты `Ingredient`, извлекать их по одному и все сразу. Определение этого класса показано в листинге 13.7.

Листинг 13.7 Тестирование репозитория Spring Data R2DBC

```
package tacos.data;

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

import java.util.ArrayList;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.r2dbc.DataR2dbcTest;

import reactor.core.publisher.Flux;
import reactor.test.StepVerifier;
import tacos.Ingredient;
import tacos.Ingredient.Type;

@DataR2dbcTest
public class IngredientRepositoryTest {

    @Autowired
    IngredientRepository ingredientRepo;

    @BeforeEach
    public void setup() {
        Flux<Ingredient> deleteAndInsert = ingredientRepo.deleteAll()
            .thenMany(ingredientRepo.saveAll(
                Flux.just(
                    new Ingredient("FLT0", "Flour Tortilla", Type.WRAP),
                    new Ingredient("GRBF", "Ground Beef", Type.PROTEIN),
                    new Ingredient("CHED", "Cheddar Cheese", Type.CHEESE)
                )));
    }
}
```

```

        StepVerifier.create(deleteAndInsert)
            .expectNextCount(3)
            .verifyComplete();
    }

    @Test
    public void shouldSaveAndFetchIngredients() {

        StepVerifier.create(ingredientRepo.findAll())
            .recordWith(ArrayList::new)
            .thenConsumeWhile(x -> true)
            .consumeRecordedWith(ingredients -> {
                assertThat(ingredients).hasSize(3);
                assertThat(ingredients).contains(
                    new Ingredient("FLTO", "Flour Tortilla", Type.WRAP));
                assertThat(ingredients).contains(
                    new Ingredient("GRBF", "Ground Beef", Type.PROTEIN));
                assertThat(ingredients).contains(
                    new Ingredient("CHED", "Cheddar Cheese", Type.CHEESE));
            })
            .verifyComplete();

        StepVerifier.create(ingredientRepo.findBySlug("FLTO"))
            .assertNext(ingredient -> {
                ingredient.equals(new Ingredient("FLTO", "Flour Tortilla",
                    Type.WRAP));
            });
    }
}

```

Метод `shouldSaveAndFetchIngredients()` создает поток `Flux` для тестовых объектов `Ingredient`. Затем применяет к нему операцию `flatMap()`, чтобы сохранить каждый ингредиент с помощью метода `save()` во внедренный `IngredientRepository`. Вызов `subscribe()` запускает передачу данных через `Flux`, и объекты `Ingredient` сохраняются в репозиторий.

Затем из потока `Mono<Ingredient>`, возвращаемого методом `findBySlug()` репозитория, создается объект `StepVerifier`. В потоке `Mono<Ingredient>` должен находиться один экземпляр `Ingredient` с кратким названием "FLTO" в свойстве `slug`, что и проверяет метод `assertNext()`.

Наконец, еще один `StepVerifier` создается из `Flux<Ingredient>`, возвращаемого методом `findAll()` репозитория. Он проверяет каждый экземпляр `Ingredient`, извлекаемый из этого потока, на совпадение с тремя объектами `Ingredient`, сохранившимися в репозиторий в начале тестирования. И, так же как в случае с другим `StepVerifier`, вызов `verifyComplete()` проверяет исчерпание потока.

Мы рассмотрели только тестирование `IngredientRepository`, однако этот подход можно использовать для тестирования любого репозитория, созданного Spring Data R2DBC.

Итак, мы определили типы данных предметной области и соответствующие им репозитории. Написали тест, проверяющий их работоспособность. Теперь можем использовать их. Но эти репозитории в текущем их виде делают сохранение TacoOrder неудобным, потому что для этого нужно сначала создать и сохранить объекты Taco, являющиеся частью заказа, а затем сохранить объект TacoOrder, ссылающийся на дочерние объекты Taco. Кроме того, при чтении TacoOrder мы получим лишь набор числовых идентификаторов объектов Taco, а не сами объекты.

Было бы неплохо иметь возможность сохранять TacoOrder как корень агрегата и вместе с ним его дочерние объекты Taco. Также было бы неплохо иметь возможность извлекать заказы TacoOrder вместе с полностью определенными объектами Taco. Для этого определим класс, который будет размещаться перед OrderRepository и TacoRepository и имитировать поведение хранилища OrderRepository из главы 3.

13.1.4 Определение службы управления агрегатами в OrderRepository

Чтобы реализовать совместное сохранение объектов TacoOrder и Taco, где TacoOrder будет играть роль корня агрегата, сначала добавим в класс TacoOrder свойство для хранения коллекции Taco (листинг 13.8).

Листинг 13.8 Добавление в класс TacoOrder свойства для хранения коллекции Taco

```
@Data  
public class TacoOrder {  
  
    ...  
  
    @Transient  
    private transient List<Taco> tacos = new ArrayList<>();  
  
    public void addTaco(Taco taco) {  
        this.tacos.add(taco);  
        if (taco.getId() != null) {  
            this.tacoIds.add(taco.getId());  
        }  
    }  
}
```

Кроме нового свойства типа `List<Taco>` и с именем `tacos` добавим также в класс TacoOrder метод `addTaco()`, включающий заданный объект Taco в этот список (и его идентификатор `id` в множество `tacoIds`).

Обратите внимание, что свойство `tacos` снабжено аннотацией `@Transient` (и отмечено ключевым словом `transient`). Так мы указыва-

ем фреймворку Spring Data R2DBC, что он не должен пытаться сохранить это свойство. Если опустить аннотацию `@Transient`, фреймворк Spring Data R2DBC попытается сохранить свойство в репозиторий, что приведет к ошибке из-за отсутствия поддержки таких отношений.

При сохранении `TacoOrder` в базу данных будет записано только свойство `tacoIds`, а свойство `tacos` – проигнорировано. Тем не менее теперь в классе `TacoOrder` есть место для хранения объектов `Taco`. Мы используем его для сохранения объектов `Taco` вместе с `TacoOrder` и для чтения объектов `Taco` при извлечении `TacoOrder`.

Теперь создадим компонент службы, который будет сохранять и извлекать объекты `TacoOrder` вместе с соответствующими объектами `Taco`. Начнем с сохранения `TacoOrder`. Эту операцию выполняет метод `save()` в классе `TacoOrderAggregateService` (листинг 13.9).

Листинг 13.9 Сохранение экземпляров `TacoOrder` и `Taco` как агрегата

```
package tacos.web.api;

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

import org.springframework.stereotype.Service;

import lombok.RequiredArgsConstructor;
import reactor.core.publisher.Mono;
import tacos.Taco;
import tacos.TacoOrder;
import tacos.data.OrderRepository;
import tacos.data.TacoRepository;

@Service
@RequiredArgsConstructor
public class TacoOrderAggregateService {

    private final TacoRepository tacoRepo;
    private final OrderRepository orderRepo;

    public Mono<TacoOrder> save(TacoOrder tacoOrder) {
        return Mono.just(tacoOrder)
            .flatMap(order -> {
                List<Taco> tacos = order.getTacos();
                order.setTacos(new ArrayList<>());
                return tacoRepo.saveAll(tacos)
                    .map(taco -> {
                        order.addTaco(taco);
                        return order;
                    }).last();
            })
            .flatMap(orderRepo::save);
    }
}
```

Хотя в листинге 13.9 не так много кода, метод `save()` выполняет множество действий, требующих пояснения. Во-первых, `TacoOrder`, полученный в параметре, заключается в поток `Mono` вызовом `Mono.just()`. Это позволяет работать с заказом в остальной части метода `save()` как с реактивным типом.

Далее к только что созданному потоку `Mono<TacoOrder>` применяется операция `flatMap()`. Операции `map()` и `flatMap()` используются для преобразования объектов данных, проходящего через `Mono` или `Flux`, но поскольку операции, выполняемые в ходе преобразования, дают в результате `Mono<TacoOrder>`, метод `flatMap()` гарантирует, что после преобразования мы получим `Mono<TacoOrder>`, а не `Mono<Mono<TacoOrder>>`, как если бы мы использовали операцию `map()`.

Цель преобразования состоит в том, чтобы гарантировать сохранение дочерних объектов `Taco` вместе с родительским объектом `TacoOrder`. Идентификатор `id` каждого объекта `Taco`, связанного с новым объектом `TacoOrder`, изначально равен нулю, и мы не узнаем истинные значения идентификаторов, пока не сохраним объекты `Taco`.

После извлечения `List<Taco>` из `TacoOrder` свойству `tacos` присваивается пустой список. Мы заполним этот список новыми объектами `Taco` с их идентификаторами, назначеными при сохранении.

Вызов метода `saveAll()` внедренного компонента `TacoRepository` сохраняет все объекты `Taco` и возвращает поток `Flux<Taco>`, который мы затем перебираем в цикле с помощью метода `map()`. В данном случае операция преобразования вторична, а главная ее задача – добавить каждый объект `Taco` обратно в `TacoOrder`. Но чтобы убедиться, что именно `TacoOrder`, а не `Taco` попадает в возвращаемый поток `Flux`, операция `map()` возвращает `TacoOrder` вместо `Taco`. Вызов `last()` гарантирует отсутствие повторяющихся объектов `TacoOrder` (по одному для каждого объекта `Taco`).

К этому моменту все объекты `Taco` должны были быть сохранены и помещены обратно в родительский объект `TacoOrder` вместе с назначенными им идентификаторами. Осталось только сохранить `TacoOrder`, что и делает последний вызов `flatMap()`. Мы вновь выбрали метод `flatMap()`, чтобы гарантировать, что `Mono<TacoOrder>`, возвращаемый вызовом `OrderRepository.save()`, не будет заключен в другой поток `Mono`. Нам нужно, чтобы наш метод `save()` возвращал `Mono<TacoOrder>`, а не `Mono<Mono<TacoOrder>>`.

Теперь рассмотрим метод, извлекающий `TacoOrder` по его идентификатору и воссоздающий все дочерние объекты `Taco`. В листинге 13.10 показан новый метод `findById()`, решающий эту задачу.

Листинг 13.10 Извлечение экземпляров `TacoOrder` и `Taco` как агрегата

```
public Mono<TacoOrder> findById(Long id) {  
    return orderRepo  
        .findById(id)  
        .flatMap(order -> {
```

```

        return tacoRepo.findAllById(order.getTacoIds())
            .map(taco -> {
                order.addTaco(taco);
                return order;
            }).last();
    });
}

```

Метод `findById()` получился немного короче метода `save()`, но от этого он не стал менее интересным.

Первое, что должен сделать этот метод, – получить `TacoOrder`. Вызов `findById()` репозитория `OrderRepository` возвращает поток `Mono<TacoOrder>`, который затем преобразуется из потока с экземпляром `TacoOrder`, содержащим только числовые идентификаторы дочерних `Taco`, в поток с экземпляром `TacoOrder`, включающим полноценные объекты `Taco`.

Лямбда-выражение, переданное методу `flatMap()`, вызывает метод `TacoRepository.findAllById()`, чтобы извлечь сразу все объекты `Taco`, на которые ссылается свойство `tacoIds`. В результате мы получаем поток `Flux<Taco>` и перебираем его в цикле с помощью `map()`, добавляя каждый объект `Taco` в родительский `TacoOrder`, подобно тому, как мы делали это в методе `save()` после сохранения всех объектов `Taco` с помощью `saveAll()`.

И снова операция `map()` используется скорее для перебора объектов `Taco`, чем для преобразования. Но лямбда-выражение, что передается операции `map()`, каждый раз возвращает родительский `TacoOrder`, поэтому мы получаем `Flux<TacoOrder>` вместо `Flux<Taco>`. Вызов `last()` извлекает последнюю запись в этом потоке и возвращает `Mono<TacoOrder>`, который затем возвращается методом `findById()`.

Код в методах `save()` и `findById()` может показаться запутанным тем, кто еще не освоился с реактивным мышлением. Реактивное программирование требует определенного навыка и поначалу может сбивать с толку, но когда вы освоите его, оно будет казаться вам довольно элегантным.

Так же как в случае с любым кодом, и особенно с кодом, который кажется запутанным, таким как код в `TacoOrderAggregateService`, рекомендуется писать тесты, чтобы убедиться, что он работает должным образом. Тест также послужит примером использования `TacoOrderAggregateService`. В листинге 13.11 показан тест для `TacoOrderAggregateService`.

Листинг 13.11 Тестирование службы `TacoOrderAggregateService`

```

package tacos.web.api;

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

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

```

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.r2dbc.DataR2dbcTest;
import org.springframework.test.annotation.DirtiesContext;

import reactor.test.StepVerifier;
import tacos.Taco;
import tacos.TacoOrder;
import tacos.data.OrderRepository;
import tacos.data.TacoRepository;

@DataR2dbcTest
@DirtiesContext
public class TacoOrderAggregateServiceTests {

    @Autowired
    TacoRepository tacoRepo;

    @Autowired
    OrderRepository orderRepo;

    TacoOrderAggregateService service;

    @BeforeEach
    public void setup() {
        this.service = new TacoOrderAggregateService(tacoRepo, orderRepo);
    }

    @Test
    public void shouldSaveAndFetchOrders() {
        TacoOrder newOrder = new TacoOrder();
        newOrder.setDeliveryName("Test Customer");
        newOrder.setDeliveryStreet("1234 North Street");
        newOrder.setDeliveryCity("Notrees");
        newOrder.setDeliveryState("TX");
        newOrder.setDeliveryZip("79759");
        newOrder.setCcNumber("4111111111111111");
        newOrder.setCcExpiration("12/24");
        newOrder.setCcCVV("123");

        newOrder.addTaco(new Taco("Test Taco One"));
        newOrder.addTaco(new Taco("Test Taco Two"));

        StepVerifier.create(service.save(newOrder))
            .assertNext(this::assertOrder)
            .verifyComplete();

        StepVerifier.create(service.findById(1L))
            .assertNext(this::assertOrder)
            .verifyComplete();
    }

    private void assertOrder(TacoOrder savedOrder) {
        assertThat(savedOrder.getId()).isEqualTo(1L);
        assertThat(savedOrder.getDeliveryName()).isEqualTo("Test Customer");
    }
}
```

```

        assertThat(savedOrder.getDeliveryName()).isEqualTo("Test Customer");
        assertThat(savedOrder.getDeliveryStreet()).isEqualTo("1234 North Street");
        assertThat(savedOrder.getDeliveryCity()).isEqualTo("Notrees");
        assertThat(savedOrder.getDeliveryState()).isEqualTo("TX");
        assertThat(savedOrder.getDeliveryZip()).isEqualTo("79759");
        assertThat(savedOrder.getCcNumber()).isEqualTo("4111111111111111");
        assertThat(savedOrder.getCcExpiration()).isEqualTo("12/24");
        assertThat(savedOrder.getCcCVV()).isEqualTo("123");
        assertThat(savedOrder.getTacoIds()).hasSize(2);
        assertThat(savedOrder.getTacos().get(0).getId()).isEqualTo(1L);
        assertThat(savedOrder.getTacos().get(0).getName())
            .isEqualTo("Test Taco One");
        assertThat(savedOrder.getTacos().get(1).getId()).isEqualTo(2L);
        assertThat(savedOrder.getTacos().get(1).getName())
            .isEqualTo("Test Taco Two");
    }
}

```

Листинг 13.11 получился довольно длинным, однако большую его часть составляют проверки содержимого `TacoOrder` в методе `assertOrder()`. В этих проверках нет ничего интересного, поэтому сосредоточимся на других аспектах нашего теста.

Тестовый класс отмечен аннотацией `@DataR2dbcTest`, чтобы фреймворк Spring создал контекст приложения со всеми необходимыми репозиториями в виде компонентов. Чтобы определить контекст приложения, аннотация `@DataR2dbcTest` отыскивает конфигурационный класс с аннотацией `@SpringBootConfiguration`. В одномодульном проекте этой цели служит класс начальной загрузки с аннотацией `@SpringBootApplication` (реализация которой сама снабжена аннотацией `@SpringBootConfiguration`). Но в нашем проекте, состоящем из нескольких модулей, тестовый класс находится в отдельном проекте, поэтому нам понадобится простой конфигурационный класс, например такой:

```

package tacos;

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

@SpringBootConfiguration
@EnableAutoConfiguration
public class TestConfig {
}

```

Это объявление не только удовлетворяет потребность в классе с аннотацией `@SpringBootConfiguration`, но также обеспечивает автоматическую настройку, гарантирующую (кроме всего прочего) создание реализаций репозиториев.

Сам по себе тест `TacoOrderAggregateServiceTests` должен выполниться без ошибок. Но в среде разработки, использующей JVM и контексты приложений Spring совместно с тестами, запуск этого теста

наряду с другими тестами репозиториев может привести к попытке сохранить в базу данных H2 конфликтующие записи. Чтобы гарантировать очистку контекста приложения Spring перед запуском теста, здесь используется аннотация `@DirtiesContext`. В результате при каждом запуске теста будет создаваться новая и пустая база данных H2.

Метод `setup()` создает экземпляр `TacoOrderAggregateService`, используя объекты `TacoRepository` и `OrderRepository`, внедренные в тестовый класс. Ссылка на `TacoOrderAggregateService` сохраняется в переменной экземпляра, чтобы методы тестирования могли ее использовать.

Теперь мы готовы протестировать службу агрегирования. Первые несколько строк в `shouldSaveAndFetchOrders()` создают объект `TacoOrder` и добавляют в него пару тестовых объектов `Taco`. Затем `TacoOrder` сохраняется вызовом метода `save()` службы `TacoOrderAggregateService`, который возвращает поток `Mono<TacoOrder>` с сохраненным заказом. Используя `StepVerifier`, мы проверяем объект `TacoOrder` в полученном потоке `Mono` на соответствие ожиданиям, включая содержащиеся в нем дочерние объекты `Taco`.

Затем вызываем метод `findById()`, который возвращает `Mono<TacoOrder>`. Так же как в случае с вызовом `save()`, здесь используется `StepVerifier` для проверки каждого `TacoOrder` в возвращаемом потоке `Mono` (он должен быть один).

В обоих случаях вызов метода `verifyComplete()` объекта `StepVerifier` помогает убедиться, что в потоке `Mono` нет других объектов и он завершил публикацию.

Мы могли бы применить аналогичное агрегирование, чтобы гарантировать включение полностью определенных объектов `Ingredient` в объекты `Taco`, но в этом не так много смысла, учитывая, что `Ingredient` является корнем собственного агрегата, на который могут ссылаться несколько объектов `Taco`. Поэтому каждый объект `Taco` будет содержать только `Set<Long>` для ссылки на идентификаторы ингредиентов, которые затем можно извлечь отдельно из `IngredientRepository`.

Как видите, Spring Data R2DBC предоставляет возможность использовать реактивные подходы для работы с реляционными данными, однако для агрегирования сущностей может потребоваться написать дополнительный код. Но это не единственный вариант реактивных хранилищ, поддерживаемый в Spring. Давайте далее посмотрим, как работать с MongoDB, используя реактивные репозитории Spring Data.

13.2 Реактивное хранилище документов в MongoDB

В главе 4 мы использовали Spring Data MongoDB для организации хранилища в документной базе данных MongoDB. В этом разделе мы вновь вернемся к MongoDB и используем поддержку реактивного программирования в Spring Data для работы с ней.

Для начала создадим проект и добавим начальную зависимость **Spring Data Reactive MongoDB**. Собственно, так подписан флагок в приложении Initializr. Также можно добавить следующие строки в спецификацию сборки проекта Maven:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-mongodb-reactive</artifactId>
</dependency>
```

В главе 4 мы использовали для тестирования Flapdoodle – встроенную базу данных MongoDB. К сожалению, Flapdoodle не особенно хорошо работает с реактивными репозиториями, поэтому для целей тестирования вам понадобится действующая база данных Mongo, прослушивающая порт 27017.

Теперь перейдем к реализации реактивного хранилища в MongoDB. Начнем с определения типов документов, составляющих нашу предметную область.

13.2.1 Определение типов документов

Как и прежде, определим классы, составляющие предметную область нашего приложения. Эти классы мы должны снабдить аннотацией `@Document` из Spring Data MongoDB, как уже делали это в главе 4, чтобы сообщить фреймворку, что они являются документами, которые должны храниться в MongoDB. Начнем с класса `Ingredient` (листинг 13.12).

Листинг 13.12 Класс Ingredient, аннотированный для хранения в базе данных MongoDB

```
package tacos;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.AccessLevel;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Document
public class Ingredient {

    @Id
    private String id;
    private String name;
    private Type type;

    public enum Type {
```

```
WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
}

}
```

Внимательный читатель заметит, что этот класс Ingredient идентичен созданному в главе 4. Фактически классы с аннотацией @Document имеют одинаковые определения, независимо от вида хранилища – реактивного или нереактивного. Соответственно, классы Taco и TacoOrder тоже будут идентичны классам, созданным в главе 4. Но для полноты картины и чтобы не возвращаться к главе 4, я повторю их здесь.

В листинге 13.13 показан аналогично аннотированный класс Taco.

Листинг 13.13 Класс Taco, аннотированный для хранения в базе данных MongoDB

```
package tacos;

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

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;
import org.springframework.data.rest.core.annotation.RestResource;

import lombok.Data;

@Data
@RestResource(rel = "tacos", path = "tacos")
@Document
public class Taco {

    @Id
    private String id;

    @NotNull
    @Size(min = 5, message = "Name must be at least 5 characters long")
    private String name;

    private Date createdAt = new Date();

    @Size(min=1, message="You must choose at least 1 ingredient")
    private List<Ingredient> ingredients = new ArrayList<>();

    public void addIngredient(Ingredient ingredient) {
        this.ingredients.add(ingredient);
    }

}
```

Обратите внимание, что, в отличие от `Ingredient`, класс `Taco` не отмечен аннотацией `@Document`. Причина в том, что экземпляры `Taco` не будут сохраняться в виде самостоятельных документов, а только как часть корня агрегата `TacoOrder`. С другой стороны, поскольку `TacoOrder` является корнем агрегата, он имеет аннотацию `@Document`, как показано в листинге 13.14.

Листинг 13.14 Класс `TacoOrder`, аннотированный для хранения в базе данных MongoDB

```
package tacos;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.Date;
import java.util.List;

import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

import lombok.Data;

@Data
@Document
public class TacoOrder implements Serializable {

    private static final long serialVersionUID = 1L;

    @Id
    private String id;
    private Date placedAt = new Date();

    private User user;

    private String deliveryName;

    private String deliveryStreet;

    private String deliveryCity;

    private String deliveryState;

    private String deliveryZip;

    private String ccNumber;

    private String ccExpiration;

    private String ccCVV;

    private List<Taco> tacos = new ArrayList<>();

    public void addTaco(Taco taco) {
        this.tacos.add(taco);
    }

}
```

Отмечу еще раз, что классы документов для реактивных репозиториев MongoDB ничем не отличаются от аналогичных классов для нереактивных репозиториев. Как вы увидите далее, реактивные репозитории MongoDB тоже очень мало отличаются от своих нереактивных аналогов.

13.2.2 Определение реактивных репозиториев MongoDB

Теперь нам нужно определить два репозитория: один для корня агрегата `TacoOrder` и другой для `Ingredient`. Мы не будем создавать отдельный репозиторий для `Taco`, потому что объекты этого типа являются членами агрегата `TacoOrder`.

Следующий интерфейс `IngredientRepository` должен быть вам знаком:

```
package tacos.data;

import org.springframework.data.repository.reactive.ReactiveCrudRepository;
import org.springframework.web.bind.annotation.CrossOrigin;
import tacos.Ingredient;

@CrossOrigin(origins="http://localhost:8080")
public interface IngredientRepository
    extends ReactiveCrudRepository<Ingredient, String> {
}
```

Этот интерфейс `IngredientRepository` лишь чуть-чуть отличается от одноименного репозитория, который мы определили в главе 4: он наследует `ReactiveCrudRepository` вместо `CrudRepository`. И отличается от репозитория для Spring Data R2DBC только отсутствием метода `findBySlug()`.

Репозиторий `OrderRepository` тоже практически идентичен репозиторию для MongoDB, созданному в главе 4:

```
package tacos.data;

import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.reactive.ReactiveCrudRepository;

import reactor.core.publisher.Flux;
import tacos.TacoOrder;
import tacos.User;

public interface OrderRepository
    extends ReactiveCrudRepository<TacoOrder, String> {

    Flux<TacoOrder> findByUserOrderByPlacedAtDesc(
        User user, Pageable pageable);
}
```

Единственное отличие реактивного репозитория от нереактивного заключается в наследовании `ReactiveCrudRepository` или `CrudRepository`.

тогу. Однако в случае выбора `ReactiveCrudRepository` клиенты таких репозиториев должны быть готовы работать с реактивными типами, такими как `Flux` и `Mono`. Это станет очевидным, когда мы будем писать тесты для реактивных репозиториев, чем мы и займемся дальше.

13.2.3 Тестирование реактивных репозиториев MongoDB

Ключевым элементом тестирования репозиториев MongoDB является класс с аннотацией `@DataMongoTest`. Эта аннотация действует подобно аннотации `@DataR2dbcTest`, которую мы использовали выше в данной главе. Она гарантирует создание контекста приложения Spring со сгенерированными репозиториями, доступными в виде компонентов. Тестовый класс может использовать эти репозитории для настройки тестовых данных и выполнения других операций с базой данных.

Например, рассмотрим класс `IngredientRepositoryTest` в листинге 13.15, который тестирует `IngredientRepository` и проверяет возможность сохранения объектов `Ingredient` в базу данных и извлечения их из нее.

Листинг 13.15 Тестирование реактивного репозитория Mongo

```
package tacos.data;

import static org.assertj.core.api.Assertions.assertThat;
import java.util.ArrayList;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.mongo.DataMongoTest;
import reactor.core.publisher.Flux;
import reactor.test.StepVerifier;
import tacos.Ingredient;
import tacos.Ingredient.Type;

@DataMongoTest
public class IngredientRepositoryTest {

    @Autowired
    IngredientRepository ingredientRepo;

    @BeforeEach
    public void setup() {
        Flux<Ingredient> deleteAndInsert = ingredientRepo.deleteAll()
            .thenMany(ingredientRepo.saveAll(
                Flux.just(
                    new Ingredient("FLTO", "Flour Tortilla", Type.WRAP),
                    new Ingredient("GRBF", "Ground Beef", Type.PROTEIN),
                    new Ingredient("CHED", "Cheddar Cheese", Type.CHEESE)
                )));
    }
}
```

```
        StepVerifier.create(deleteAndInsert)
            .expectNextCount(3)
            .verifyComplete();
    }

    @Test
    public void shouldSaveAndFetchIngredients() {

        StepVerifier.create(ingredientRepo.findAll())
            .recordWith(ArrayList::new)
            .thenConsumeWhile(x -> true)
            .consumeRecordedWith(ingredients -> {
                assertThat(ingredients).hasSize(3);
                assertThat(ingredients).contains(
                    new Ingredient("FLTO", "Flour Tortilla", Type.WRAP));
                assertThat(ingredients).contains(
                    new Ingredient("GRBF", "Ground Beef", Type.PROTEIN));
                assertThat(ingredients).contains(
                    new Ingredient("CHED", "Cheddar Cheese", Type.CHEESE));
            })
            .verifyComplete();

        StepVerifier.create(ingredientRepo.findById("FLTO"))
            .assertNext(ingredient -> {
                ingredient.equals(new Ingredient("FLTO", "Flour Tortilla",
                                                Type.WRAP));
            });
    }
}
```

Этот тест похож на тест репозитория R2DBC, который мы написали выше в этой главе, но все же имеет некоторые отличия. Он начинается с записи в базу данных трех объектов `Ingredient`. Затем использует два экземпляра `StepVerifier` для проверки возможности чтения данных из репозитория, сначала как множества всех объектов `Ingredient`, а затем одного экземпляра `Ingredient` по его идентификатору.

Кроме того, так же как в предыдущем тесте для R2DBC, аннотация `@DataMongoTest` будет искать класс с аннотацией `@SpringBootConfiguration`, чтобы создать контекст приложения. Далее выполняются те же проверки, что и в предыдущем teste.

Основное отличие этого теста в том, что первый `StepVerifier` выбирает все объекты `Ingredient` в список `ArrayList` и затем проверяет его содержимое. Метод `findAll()` не гарантирует получение результатов в каком-то определенном порядке, что не позволяет использовать методы `assertNext()` или `expectNext()`. Собрав все полученные объекты `Ingredient` в список, можно проверить присутствие всех трех объектов независимо от порядка их следования.

Тест для `OrderRepository` выглядит очень похоже, как показано в листинге 13.16.

Листинг 13.16 Тестирование репозитория OrderRepository для Mongo

```
package tacos.data;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.mongo.DataMongoTest;

import reactor.test.StepVerifier;
import tacos.Ingredient;
import tacos.Taco;
import tacos.TacoOrder;
import tacos.Ingredient.Type;

@DataMongoTest
public class OrderRepositoryTest {

    @Autowired
    OrderRepository orderRepo;

    @BeforeEach
    public void setup() {
        orderRepo.deleteAll().subscribe();
    }

    @Test
    public void shouldSaveAndFetchOrders() {
        TacoOrder order = createOrder();

        StepVerifier
            .create(orderRepo.save(order))
            .expectNext(order)
            .verifyComplete();

        StepVerifier
            .create(orderRepo.findById(order.getId()))
            .expectNext(order)
            .verifyComplete();

        StepVerifier
            .create(orderRepo.findAll())
            .expectNext(order)
            .verifyComplete();
    }

    private TacoOrder createOrder() {
        TacoOrder order = new TacoOrder();
        ...
        return order;
    }
}
```

Первым делом метод `shouldSaveAndFetchOrders()` создает заказ со сведениями о покупателе, платежной информацией и парой тако (я опустил содержимое метода `createOrder()` для краткости). Затем с помощью `StepVerifier` сохраняет получившийся объект `TacoOrder` и сравнивает его с объектом, который возвращается методом `save()`. После этого предпринимается попытка получить заказ по его идентификатору и проверяется получение полного объекта `TacoOrder`. Наконец, `shouldSaveAndFetchOrders()` извлекает все объекты `TacoOrder` (должен быть только один заказ) и сравнивает с ожидаемым `TacoOrder`.

Как упоминалось выше, для запуска этого теста вам понадобится действующий сервер MongoDB, прослушивающий порт 27017; Flapdoodle –строенная реализация MongoDB – не особенно хорошо работает с реактивными репозиториями. Если на вашем компьютере установлено программное обеспечение Docker, то вы можете легко запустить сервер MongoDB, прослушивающий порт 27017, следующей командой:

```
$ docker run -p27017:27017 mongo
```

Возможны также другие способы установки MongoDB. За более подробной информацией обращайтесь к документации по адресу <https://www.mongodb.com/>.

Теперь, узнав, как создавать реактивные репозитории для R2DBC и MongoDB, рассмотрим еще один вариант применения Spring Data для организации реактивного хранилища: Cassandra.

13.3 Реактивное хранилище данных в Cassandra

Чтобы получить возможность реактивного доступа к базе данных Cassandra, нужно добавить следующую начальную зависимость в спецификацию сборки проекта взамен любых зависимостей Mongo или R2DBC, которые использовались выше.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-cassandra-reactive</artifactId>
</dependency>
```

Затем нужно определить некоторые детали о пространстве ключей Cassandra и управлении схемой. Для этого добавьте следующие строки в файл `application.yml`:

```
spring:
  data:
    rest:
      base-path: /data-api
    cassandra:
```

```
keyspace-name: tacocloud
schema-action: recreate
local-datacenter: datacenter1
```

Это та же конфигурация в формате YAML, которая использовалась в главе 4 для организации нереактивных репозиториев Cassandra. Особое внимание следует обратить на имя пространства ключей в свойстве `keyspace-name` – пространство ключей с таким же именем обязательно должно быть создано в кластере Cassandra.

Вам также понадобится кластер Cassandra, действующий на вашем локальном компьютере и прослушивающий порт 9042. Самый простой способ запустить такой кластер – использовать Docker:

```
$ docker network create cassandra-net
$ docker run --name my-cassandra --network cassandra-net \
    -p 9042:9042 -d cassandra:latest
```

Если ваш кластер Cassandra действует на другой машине или прослушивает другой порт, то определите свойства `contact-points` и `port` в файле `application.yml`, как показано в главе 4. Чтобы создать пространство ключей, запустите оболочку CQL и используйте команду `create keyspace`:

```
$ docker run -it --network cassandra-net --rm cassandra cqlsh my-cassandra
cqlsh> create keyspace tacocloud
WITH replication = {'class': 'SimpleStrategy', 'replication_factor' : 1};
```

Теперь, имея действующий кластер Cassandra, новое пространство ключей `tacocloud` и добавив начальную зависимость Spring Data Cassandra Reactive в проект, можно приступить к определению классов предметной области.

13.3.1 Определение классов предметной области для Cassandra

Так же как при использовании Mongo, выбор между реактивным и нереактивным доступом к Cassandra абсолютно не влияет на определения классов предметной области. Классы `Ingredient`, `Taco` и `TacoOrder`, которые мы будем использовать далее, совершенно идентичны созданным в главе 4. В листинге 13.17 показан класс `Ingredient`, аннотированный для работы с Cassandra.

Листинг 13.17 Класс Ingredient, аннотированный для работы с Cassandra

```
package tacos;

import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.mapping.Table;

import lombok.AccessLevel;
```

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor(access=AccessLevel.PRIVATE, force=true)
@Table("ingredients")
public class Ingredient {

    @PrimaryKey
    private String id;
    private String name;
    private Type type;

    public enum Type {
        WRAP, PROTEIN, VEGGIES, CHEESE, SAUCE
    }

}
```

Класс Taco определен с использованием аналогичных аннотаций, как показано в листинге 13.18.

Листинг 13.18 Класс Taco, аннотированный для работы с Cassandra

```
package tacos;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.UUID;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

import org.springframework.data.cassandra.core.cql.Ordering;
import org.springframework.data.cassandra.core.cql.PrimaryKeyType;
import org.springframework.data.cassandra.core.mapping.Column;
import org.springframework.data.cassandra.core.mapping.PrimaryKeyColumn;
import org.springframework.data.cassandra.core.mapping.Table;
import org.springframework.data.rest.core.annotation.RestResource;

import com.datastax.oss.driver.api.core.uuid.Uuids;

import lombok.Data;

@Data
@RestResource(rel = "tacos", path = "tacos")
@Table("tacos")
public class Taco {

    @PrimaryKeyColumn(type=PrimaryKeyType.PARTITIONED)
    private UUID id = Uuids.timeBased();

    @NotNull
```

```

@Size(min = 5, message = "Name must be at least 5 characters long")
private String name;

@PrimaryKeyColumn(type=PrimaryKeyType.CLUSTERED,
                  ordering=Ordering.DESCENDING)
private Date createdAt = new Date();

@Size(min=1, message="You must choose at least 1 ingredient")
@Column("ingredients")
private List<IngredientUDT> ingredients = new ArrayList<>();

public void addIngredient(Ingredient ingredient) {
    this.ingredients.add(new IngredientUDT(ingredient.getName(),
                                             ingredient.getType()));
}

}

```

Поскольку объекты Taco ссылаются на соответствующие объекты Ingredient через пользовательский тип, нам также понадобится класс IngredientUDT, показанный в листинге 13.19.

Листинг 13.19 Пользовательский тип IngredientUDT для работы с Cassandra

```

package tacos;

import org.springframework.data.cassandra.core.mapping.UserDefinedType;

import lombok.AccessLevel;
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@NoArgsConstructor
@UserDefinedType("ingredient")
public class IngredientUDT {

    private String name;
    private Ingredient.Type type;
}

```

В листинге 13.20 показан последний из трех классов предметной области – TacoOrder, аннотированный для работы с Cassandra.

Листинг 13.20 Класс TacoOrder, аннотированный для работы с Cassandra

```

package tacos;

import java.io.Serializable;
import java.util.ArrayList;

```

```
import java.util.Date;
import java.util.List;
import java.util.UUID;

import org.springframework.data.cassandra.core.mapping.Column;
import org.springframework.data.cassandra.core.mapping.PrimaryKey;
import org.springframework.data.cassandra.core.mapping.Table;

import com.datastax.oss.driver.api.core.uuid.Uuids;

import lombok.Data;

@Data
@Table("tacoorders")
public class TacoOrder implements Serializable {

    private static final long serialVersionUID = 1L;

    @PrimaryKey
    private UUID id = Uuids.timeBased();
    private Date placedAt = new Date();

    @Column("user")
    private UserUDT user;

    private String deliveryName;

    private String deliveryStreet;

    private String deliveryCity;

    private String deliveryState;

    private String deliveryZip;

    private String ccNumber;

    private String ccExpiration;

    private String ccCVV;

    @Column("tacos")
    private List<TacoUDT> tacos = new ArrayList<>();

    public void addTaco(Taco taco) {
        this.addTaco(new TacoUDT(taco.getName(), taco.getIngredients()));
    }

    public void addTaco(TacoUDT tacoUDT) {
        this.tacos.add(tacoUDT);
    }

}
```

Подобно объектам Taco, ссылающимся на объекты Ingredient через определяемый пользователем тип, объекты TacoOrder будут ссылаться на объекты Taco через класс TacoUDT, который показан в листинге 13.21.

Листинг 13.21 Пользовательский тип TacoUDT для работы с Cassandra

```
package tacos;

import java.util.List;

import org.springframework.data.cassandra.core.mapping.UserDefinedType;

import lombok.Data;

@Data
@UserDefinedType("taco")
public class TacoUDT {

    private final String name;
    private final List<IngredientUDT> ingredients;

}
```

Как уже говорилось, все эти классы совершенно идентичны своим нереактивным аналогам. Я повторил их определения здесь только ради того, чтобы вам не пришлось листать книгу на 11 глав назад.

Теперь определим репозитории, сохраняющие эти объекты.

13.3.2 Создание реактивных репозиториев Cassandra

Уверен, вы ожидаете, что реактивные репозитории Cassandra будут очень похожи на эквивалентные нереактивные репозитории. И это действительно так! Вы уже понимаете, что везде, где это возможно, Spring Data старается обеспечить единообразие модели программирования независимо от типа репозиториев.

Возможно, вы уже догадались, что единственное отличие, превращающее нереактивный репозиторий в реактивный, – это наследуемый интерфейс, в данном случае `ReactiveCrudRepository`, как показано в примере определения интерфейса `IngredientRepository`:

```
package tacos.data;

import org.springframework.data.repository.reactive.ReactiveCrudRepository;

import tacos.Ingredient;

public interface IngredientRepository
    extends ReactiveCrudRepository<Ingredient, String> {
}
```

То же верно для интерфейса `OrderRepository`:

```
package tacos.data;

import java.util.UUID;
```

```
import org.springframework.data.domain.Pageable;
import org.springframework.data.repository.reactive.ReactiveCrudRepository;

import reactor.core.publisher.Flux;
import tacos.TacoOrder;
import tacos.User;

public interface OrderRepository
    extends ReactiveCrudRepository<TacoOrder, UUID> {

    Flux<TacoOrder> findByUserOrderByPlacedAtDesc(
        User user, Pageable pageable);
}
```

На самом деле эти репозитории не только напоминают свои не-реактивные аналоги, но почти не отличаются от репозиториев MongoDB, описанных выше в данной главе. Репозиторий для Cassandra использует UUID в качестве типа идентификатора для TacoOrder вместо String, но в остальном они практически идентичны. Это еще раз демонстрирует приверженность Spring Data к единообразию.

Давайте завершим наше знакомство с созданием реактивных репозиториев Cassandra, написав пару тестов, помогающих убедиться в их работоспособности.

13.3.3 Тестирование реактивных репозиториев Cassandra

Теперь вас уже не должно удивлять сходство приемов тестирования реактивных репозиториев Cassandra и MongoDB. Например, взгляните на IngredientRepositoryTest в листинге 13.22 и проверьте себя – сможете ли вы определить, чем он отличается от листинга 13.15.

Листинг 13.22 Тестирование реактивного репозитория IngredientRepository для Cassandra

```
package tacos.data;

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

import java.util.ArrayList;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.autoconfigure.data.cassandra
    .DataCassandraTest;

import reactor.core.publisher.Flux;
import reactor.test.StepVerifier;
import tacos.Ingredient;
import tacos.Ingredient.Type;

@DataCassandraTest
```

```

public class IngredientRepositoryTest {

    @Autowired
    IngredientRepository ingredientRepo;

    @BeforeEach
    public void setup() {
        Flux<Ingredient> deleteAndInsert = ingredientRepo.deleteAll()
            .thenMany(ingredientRepo.saveAll(
                Flux.just(
                    new Ingredient("FLTO", "Flour Tortilla", Type.WRAP),
                    new Ingredient("GRBF", "Ground Beef", Type.PROTEIN),
                    new Ingredient("CHED", "Cheddar Cheese", Type.CHEESE)
                )));
        StepVerifier.create(deleteAndInsert)
            .expectNextCount(3)
            .verifyComplete();
    }

    @Test
    public void shouldSaveAndFetchIngredients() {

        StepVerifier.create(ingredientRepo.findAll())
            .recordWith(ArrayList::new)
            .thenConsumeWhile(x -> true)
            .consumeRecordedWith(ingredients -> {
                assertThat(ingredients).hasSize(3);
                assertThat(ingredients).contains(
                    new Ingredient("FLTO", "Flour Tortilla", Type.WRAP));
                assertThat(ingredients).contains(
                    new Ingredient("GRBF", "Ground Beef", Type.PROTEIN));
                assertThat(ingredients).contains(
                    new Ingredient("CHED", "Cheddar Cheese", Type.CHEESE));
            })
            .verifyComplete();

        StepVerifier.create(ingredientRepo.findById("FLTO"))
            .assertNext(ingredient -> {
                ingredient.equals(new Ingredient("FLTO", "Flour Tortilla",
                    Type.WRAP));
            });
    }
}

```

Заметили разницу? Там, где в версии для MongoDB использовалась аннотация `@DataMongoTest`, в версии для Cassandra используется аннотация `@DataCassandraTest`. И все! В остальном тесты абсолютно идентичны.

То же верно и для `OrderRepositoryTest`. Нужно лишь заменить `@DataMongoTest` на `@DataCassandraTest`, а весь остальной код можно оставить прежним:

```
@DataCassandraTest  
public class OrderRepositoryTest {  
    ...  
}
```

Как видите, единство в различных проектах Spring Data распространяется даже на создание тестов. Это позволяет легко переключаться между разными механизмами хранения данных, не задумываясь о различиях между ними.

Итоги

- Spring Data поддерживает реактивную модель взаимодействий с различными типами баз данных, включая реляционные (R2DBC), MongoDB и Cassandra.
- Spring Data R2DBC предлагает реактивную модель доступа к реляционному хранилищу, но она пока не имеет прямой поддержки отношений между классами предметной области.
- Из-за отсутствия прямой поддержки отношений репозитории Spring Data R2DBC требуют использовать иной подход к проектированию типов и таблиц баз данных.
- Spring Data MongoDB и Spring Data Cassandra предлагают практически идентичные модели программирования для разработки реактивных репозиториев баз данных MongoDB и Cassandra.
- Используя тестовые аннотации Spring Data вместе с объектом `StepVerifier`, можно тестировать реактивные репозитории, автоматически создаваемые в контексте приложения Spring.

14 RSocket

В этой главе рассматриваются следующие темы:

- реактивные сетевые взаимодействия с RSocket;
- четыре коммуникационные модели RSocket;
- передача по протоколу RSocket через WebSocket.

Во времена до появления телефонов и современной электроники лучшим способом общения с друзьями и родственниками, живущими далеко, были обычные письма, пересылаемые по почте. Это была не-быстрая форма общения, порой проходило несколько дней или даже недель, прежде чем вы получали ответ, но это была единственная доступная возможность.

Благодаря Александру Грэму Беллу (Alexander Graham Bell), изобретателю телефона, у нас появился новый способ общения с друзьями и родственниками, обеспечивший мгновенную связь и превративший письмо почти в утраченное искусство.

Что касается связи между приложениями, наибольшее распространение получила модель «запрос–ответ», предлагаемая службами HTTP и REST. Но она имеет определенные ограничения. Подобно написанию писем, модель «запрос–ответ» предполагает отправку сообщения и последующее ожидание ответа. Эта модель не поддерживает асинхронную связь, когда сервер может посыпать поток ответов, и не обеспечивает открытый двунаправленный канал, по которому клиент и сервер могут снова и снова пересыпать данные друг другу.

В этой главе мы рассмотрим RSocket, относительно новый протокол взаимодействий между приложениями, который позволяет обмениваться не только запросами и ответами. И поскольку он носит реактивный характер, он намного эффективнее блокирующих HTTP-запросов.

Попутно мы посмотрим, как организовать взаимодействия по протоколу RSocket в Spring. Но сначала рассмотрим протокол RSocket в общем, чтобы увидеть его отличия от HTTP.

14.1 Введение в RSocket

RSocket (<https://rsocket.io/>) – это прикладной двоичный протокол, действующий асинхронно и основанный на Reactive Streams. Иначе говоря, RSocket обеспечивает возможность асинхронных взаимодействий между приложениями на основе реактивной модели, совместимой с реактивными типами, такими как Flux и Mono, с которыми мы познакомились в главе 12.

Будучи альтернативой HTTP, этот протокол более гибкий и реализует четыре модели взаимодействий: «запрос–ответ», «запрос–поток», «запустил и забыл» и «канал».

Запрос–ответ – наиболее знакомая модель взаимодействий, имитирующая работу типичной модели взаимодействий через HTTP. В модели «запрос–ответ» клиент отправляет серверу один запрос, а сервер отвечает одним ответом. Принцип действия этой модели показан на рис. 14.1, где используется тип Mono для определения запроса и ответа.

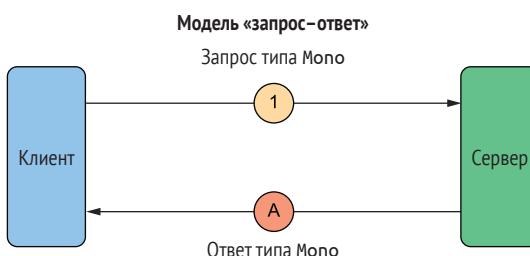


Рис. 14.1 Модель взаимодействий «запрос–ответ» по протоколу RSocket

На первый взгляд может показаться, что модель «запрос–ответ» эквивалентна модели взаимодействий, предлагаемой протоколом HTTP, однако важно понимать, что RSocket является принципиально не блокирующим протоколом и основан на реактивных типах. Конечно, клиент по-прежнему вынужден ждать ответа от сервера, но при этом он не блокируется и может своевременно реагировать на ответы, что позволяет более эффективно использовать параллельные потоки выполнения.

Модель **запрос–поток** напоминает модель «запрос–ответ», за исключением того, что в ответ на один запрос клиента сервер отвечает потоком значений. На рис. 14.2 показан принцип действия модели «запрос–поток», где для представления запроса используется тип Mono, а для ответов – тип Flux.

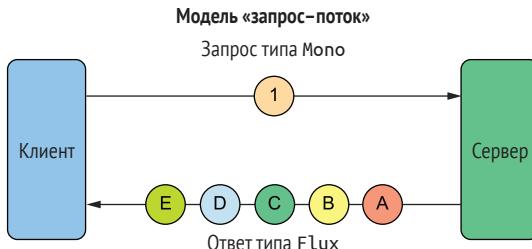


Рис. 14.2 Модель взаимодействий «запрос–поток» по протоколу RSocket

В некоторых случаях клиенту может потребоваться отправить данные на сервер, а что ответит сервер – совершенно не важно. Для таких случаев RSocket предлагает модель «запустил и забыл», принцип действия которой показан на рис. 14.3.

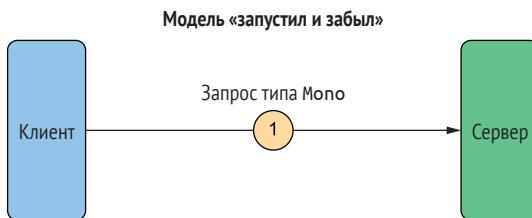


Рис. 14.3 Модель взаимодействий «запустил и забыл» по протоколу RSocket

В модели «запустил и забыл» клиент посылает запрос серверу, но сервер не отправляет ответ.

Наконец, наиболее гибкой из моделей взаимодействий в RSocket является модель «канал». В этой модели клиент открывает двунаправленный канал с сервером, и оба получают возможность отправлять данные друг другу в любое время. На рис. 14.4 показан принцип действия модели «канал».

Протокол RSocket поддерживается различными языками и платформами, включая Java, JavaScript, Kotlin, .NET, Go и C++¹. Последние версии Spring предлагают превосходную поддержку RSocket,

¹ Это сильно сокращенный список языков, полный перечень вы найдете на веб-сайте RSocket, но имейте в виду, что могут существовать реализации RSocket для других языков, созданные сообществами.

упрощающую создание серверов и клиентов с помощью знакомых идиом Spring.

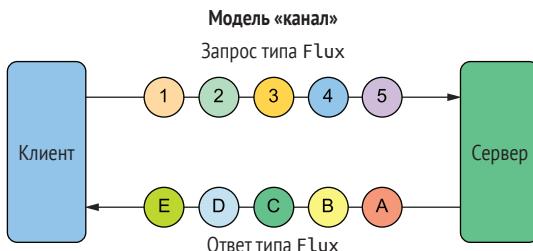


Рис. 14.4 Модель взаимодействий «канал» по протоколу RSocket

А теперь давайте посмотрим, как создавать серверы и клиенты RSocket для каждой из четырех моделей.

14.2 Создание простого сервера и клиента RSocket

Spring предлагает превосходную поддержку взаимодействий по протоколу RSocket с использованием всех четырех моделей. Чтобы начать использовать RSocket, нужно добавить начальную зависимость Spring Boot RSocket в спецификацию сборки проекта. Вот как выглядит начальная зависимость RSocket в POM-файле для Maven.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-rsocket</artifactId>
</dependency>
```

Эту зависимость необходимо подключать и в серверных, и в клиентских приложениях, участвующих в обмене данными через RSocket.

ПРИМЕЧАНИЕ При настройке зависимостей в Spring Initializr можно увидеть аналогичную зависимость WebSocket. Названия протоколов RSocket и WebSocket очень похожи, и даже есть возможность использовать WebSocket в качестве транспортного протокола для RSocket (мы рассмотрим эту возможность в данной главе), но вы должны выбрать именно зависимость RSocket, а не WebSocket.

Затем нужно выбрать модель взаимодействий. Не существует четких критерииев, которыми можно было бы руководствоваться в каждой конкретной ситуации, поэтому вы должны будете взвесить свой выбор, опираясь на желаемое поведение вашего приложения. Одна-

ко, как будет показано в следующих нескольких примерах, подходы к реализации разных моделей не сильно отличаются, поэтому вы легко сможете переключиться на другую модель, если первоначальный выбор окажется ошибочным.

Давайте посмотрим, как создать сервер и клиент RSocket в Spring, используя каждую из моделей. Поскольку все модели взаимодействий, поддерживаемые протоколом RSocket, имеют свои особенности и лучше приспособлены для разных сценариев использования, мы отложим приложение Taco Cloud в сторону и посмотрим, как применять RSocket в различных ситуациях. Начнем с применения модели «запрос–ответ».

14.2.1 Реализация модели «запрос–ответ»

Чтобы создать сервер RSocket в Spring, достаточно определить класс контроллера, почти так же, как в веб-приложении или в службе REST. Контроллер в листинге 14.2 является примером службы RSocket, обрабатывающей приветствие клиента и отвечающей своим приветствием.

Листинг 14.2 Простой сервер RSocket, реализующий модель «запрос–ответ»

```
package rssocket;

import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.stereotype.Controller;

import lombok.extern.slf4j.Slf4j;
import reactor.core.publisher.Mono;

@Controller
@Slf4j
public class GreetingController {

    @MessageMapping("greeting")
    public Mono<String> handleGreeting(Mono<String> greetingMono) {
        return greetingMono
            .doOnNext(greeting ->
                log.info("Received a greeting: {}", greeting))
            .map(greeting -> "Hello back to you!");
    }
}
```

Как видите, ключевое отличие контроллера RSocket от веб-контроллера заключается в том, что вместо HTTP-запросов к заданной конечной точке (с помощью методов с аннотацией `@GetMapping` или `@PostMapping`) контроллер RSocket обрабатывает входящие сообщения с помощью метода с аннотацией `@MessageMapping`. В этом примере ме-

тод `handleGreeting()` вызывается, когда клиент отправляет запрос по маршруту "greeting".

Метод `handleGreeting()` получает данные из сообщения в параметре типа `Mono<String>`. В данном случае приветствие настолько простое, что для его представления достаточно типа `String`, но при необходимости входящие данные могут быть более сложного типа. Получив `Mono<String>`, метод просто фиксирует в журнале факт получения приветствия, а затем применяет операцию `map()` к потоку `Mono`, чтобы создать новый поток `Mono<String>` для передачи ответа клиенту.

Контроллеры RSocket обрабатывают маршруты не так, как пути в HTTP-запросах, и все же есть возможность так задать маршрут, что он будет выглядеть как путь, включая переменные, которые будут переданы в вызов метода обработчика. Например, взгляните на следующий вариант метода `handleGreeting()`:

```
@MessageMapping("greeting/{name}")
public Mono<String> handleGreeting(
    @DestinationVariable("name") String name,
    Mono<String> greetingMono) {
    return greetingMono
        .doOnNext(greeting ->
            log.info("Received a greeting from {} : {}", name, greeting))
        .map(greeting -> "Hello to you, too, " + name);
}
```

В этом случае маршрут, указанный в `@MessageMapping`, содержит переменную с именем "name". Она обозначается фигурными скобками так же, как переменные в HTTP URL. Метод принимает соответствующий параметр типа `String` с аннотацией `@DestinationVariable`, ссылающейся на переменную. Подобно аннотации `@PathVariable` в Spring MVC, `@DestinationVariable` используется для извлечения значения, указанного в переменной в маршруте, и его передачи в метод обработчика. Эта новая версия `handleGreeting()` использует полученное имя для создания более персонализированного приветствия.

Есть еще один аспект, о котором следует помнить при создании сервера RSocket: прослушиваемый порт. По умолчанию службы RSocket основаны на TCP и являются автономными серверами, прослушивающими определенные порты. Задать порт для службы RSocket можно с помощью конфигурационного свойства `spring.rsocket.server.port`:

```
spring:
rsocket:
server:
port: 7000
```

Свойство `spring.rsocket.server.port` служит двум целям: активизирует сервер и сообщает ему номер порта, который тот должен прослушивать. Если это свойство не задано, то Spring будет считать, что приложение действует только как клиент и не предполагает прием

каких-либо запросов. Наш пример должен работать как сервер, прослушивающий порт 7000, поэтому устанавливаем свойство `spring.rsocket.server.port`, как показано выше.

Теперь обратим свой взор на сторону клиента RSocket. В Spring клиенты RSocket реализуются с использованием `RSocketRequester`. Механизм автоконфигурации RSocket в Spring Boot автоматически создаст компонент типа `RSocketRequester.Builder` в контексте приложения. Этот компонент-построитель можно внедрить в любой другой компонент, чтобы создать экземпляр `RSocketRequester`.

Например, вот начало реализации компонента `ApplicationRunner`, в который внедряется `RSocketRequester.Builder`:

```
package rssocket;

import org.springframework.boot.ApplicationRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.rsocket.RSocketRequester;

@Configuration
@Slf4j
public class RSocketClientConfiguration {

    @Bean
    public ApplicationRunner sender(RSocketRequester.Builder requesterBuilder)
    {
        return args -> {
            RSocketRequester tcp = requesterBuilder.tcp("localhost", 7000);

            // ... послать сообщение с помощью RSocketRequester ...

        };
    }
}
```

В этом случае построитель используется для создания объекта `RSocketRequester`, который прослушивает порт 7000 на `localhost`. Этот объект `RSocketRequester` затем можно использовать для отправки сообщений на сервер.

В модели «запрос–ответ» запрос должен содержать (как минимум) маршрут и, возможно, полезные данные. Как вы помните, контроллер нашего сервера обрабатывает запросы с маршрутом "greeting" и ожидает получить строку. Кроме того, в ответ он возвращает строку. В листинге 14.3 показан полный код клиента, отправляющий приветствие на сервер и обрабатывающий ответ.

Листинг 14.3 Отправка запроса клиентом

```
RSocketRequester tcp = requesterBuilder.tcp("localhost", 7000);

// ... послать сообщение с помощью RSocketRequester ...
```

```
tcp
    .route("greeting")
    .data("Hello RSocket!")
    .retrieveMono(String.class)
    .subscribe(response -> log.info("Got a response: {}", response));
```

Этот код отправляет приветствие "Hello RSocket!" по маршруту "greeting". Обратите внимание, что он также ожидает получить в ответ Mono<String>, вызывая `retrieveMono()`. Метод `subscribe()` подписывается на возвращаемый поток Mono и обрабатывает его, записывая полученное значение в журнал.

Теперь предположим, что мы решили отправить приветствие по другому маршруту, который принимает переменную. Код на стороне клиента в этом случае действует почти так же, с той лишь разницей, что включает переменную и ее значение в маршрут, передаваемый в вызов `route()`:

```
String who = "Craig";
tcp
    .route("greeting/{name}", who)
    .data("Hello RSocket!")
    .retrieveMono(String.class)
    .subscribe(response -> log.info("Got a response: {}", response));
```

В этом случае сообщение будет отправлено с маршрутом "greeting/Craig" и обработано методом контроллера, аннотация `@Message-Mapping` которого определяет маршрут как "greeting/{name}". Мы могли бы вставить имя прямо в строку маршрута или сконструировать маршрут с помощью операции конкатенации строк, однако прием с использованием заполнителя выглядит более очевидным и менее подвержен ошибкам.

Модель «запрос–ответ», вероятно, самая простая из коммуникационных моделей, поддерживаемых RSocket, но это только начало. Давайте теперь перейдем к модели «запрос–поток», позволяющей возвращать несколько ответов.

14.2.2 Реализация модели «запрос–поток»

Не все взаимодействия заключаются в передаче одного запроса и получении одного ответа. Например, в биржевых приложениях может быть полезно запросить поток котировок для данного символа акции. В модели «запрос–ответ» клиент должен неоднократно запрашивать текущую цену акции. Но в модели «запрос–поток» клиенту достаточно запросить цену акции только один раз, а затем подписаться на поток периодических обновлений.

Для иллюстрации модели «запрос–поток» реализуем сервер и клиента, действующих по сценарию передачи котировок акций. Для начала определим объект, содержащий информацию о котировках акций. Этой цели будет служить класс `StockQuote` в листинге 14.4.

Листинг 14.4 Класс, представляющий котировки акций

```
package rsocket;

import java.math.BigDecimal;
import java.time.Instant;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class StockQuote {

    private String symbol;
    private BigDecimal price;
    private Instant timestamp;
}
```

Как видите, `StockQuote` содержит символ акции, цену и отметку времени, на которое была зафиксирована эта цена. Чтобы сократить объем кода, используем библиотеку Lombok, которая автоматически создаст конструкторы и методы доступа.

Теперь напишем контроллер для обработки запросов на получение котировок акций. Контроллер `StockQuoteController` в листинге 14.5 очень похож на `GreetingController` из предыдущего раздела.

Листинг 14.5 Контроллер RSocket для передачи потока котировок акций

```
package rsocket;

import java.math.BigDecimal;
import java.time.Duration;
import java.time.Instant;

import org.springframework.messaging.handler.annotation.DestinationVariable;
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.stereotype.Controller;

import reactor.core.publisher.Flux;

@Controller
public class StockQuoteController {

    @MessageMapping("stock/{symbol}")
    public Flux<StockQuote> getStockPrice(
        @DestinationVariable("symbol") String symbol) {
        return Flux
            .interval(Duration.ofSeconds(1))
            .map(i -> {
                BigDecimal price = BigDecimal.valueOf(Math.random() * 10);
                StockQuote quote = new StockQuote();
                quote.symbol = symbol;
                quote.price = price;
                quote.timestamp = Instant.now();
                return quote;
            });
    }
}
```

```

        return new StockQuote(symbol, price, Instant.now());
    });
}
}

```

Метод `getStockPrice()` обрабатывает входящие запросы с маршрутом "`stock/{symbol}`" и принимает символ акции из маршрута с помощью аннотации `@DestinationVariable`. Для простоты вместо поиска фактических котировок мы будем вычислять их как случайные значения (если хотите, представьте себе это как моделирование волатильности некоторых реальных акций).

Однако самое примечательное в `getStockPrice()` – он возвращает `Flux<StockQuote>` вместо `Mono<StockQuote>`. Эта особенность подсказывает фреймворку Spring, что метод `getStockPrice()` поддерживает модель «запрос–поток». Первичный поток `Flux` создается на основе таймера, срабатывающего каждую секунду, но он отображается в другой поток `Flux`, генерирующий случайную котировку `StockQuote`. Проще говоря, в ответ на один запрос метод `getStockPrice()` возвращает несколько значений, по одному каждую секунду.

Клиент службы «запрос–поток» реализуется аналогично клиенту службы «запрос–ответ». Единственное существенное отличие состоит в том, что вместо `retrieveMono()` вызывается функция `retrieveFlux()`. Вот как может выглядеть клиент службы котировок:

```

String stockSymbol = "XYZ";

RSocketRequester tcp = requesterBuilder.tcp("localhost", 7000);
tcp
    .route("stock/{symbol}", stockSymbol)
    .retrieveFlux(StockQuote.class)
    .doOnNext(stockQuote ->
        log.info(
            "Price of {} : {} (at {})",
            stockQuote.getSymbol(),
            stockQuote.getPrice(),
            stockQuote.getTimestamp()
        )
    )
    .subscribe();

```

Теперь вы знаете, как создавать серверы и клиенты RSocket, обрабатывающие одиночные и множественные ответы. А что, если у сервера нет ответа или клиенту не нужен ответ? Давайте посмотрим, как работать с коммуникативной моделью «запустил и забыл».

14.2.3 Реализация модели «запустил и забыл»

Представьте, что мы находимся в звездолете, который только что подвергся нападению вражеского корабля. Мы объявляем «боевую тревогу» по всему кораблю, чтобы перевести все службы в боевой режим.

Нам не нужно ждать ответа от корабельных компьютеров, подтверждающих переход в боевой режим, и у нас нет времени ждать и читать любые ответы в такой ситуации. Мы отправляем оповещение, а затем переходим к более важным вопросам.

Это пример модели «запустил и забыл». Конечно, мы не забываем, что находимся в боевом режиме, но, учитывая обстоятельства, нам важнее отразить нападение, чем проверять реакцию служб на объявление боевой тревоги.

Чтобы смоделировать этот сценарий, создадим сервер RSocket, который обрабатывает сообщения, но ничего не возвращает. Во-первых, нам нужно определить класс, представляющий полезную нагрузку запроса, например класс `Alert`, показанный в листинге 14.6.

Листинг 14.6 Класс, представляющий запрос объявления тревоги

```
package rsocket;

import java.time.Instant;
import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class Alert {

    private Level level;
    private String orderedBy;
    private Instant orderedAt;

    public enum Level {
        YELLOW, ORANGE, RED, BLACK
    }
}
```

Объект `Alert` задает уровень тревоги и отметку времени, когда оповещение было отправлено (определяется с типом `Instant`). И снова мы используем библиотеку Lombok, чтобы автоматически создать необходимые конструкторы и методы доступа.

На стороне сервера сообщения `Alert` будет обрабатывать контроллер `AlertController`, представленный в листинге 14.7.

Листинг 14.7 Контроллер RSocket для обработки запросов объявления тревоги

```
package rsocket;

import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.stereotype.Controller;

import lombok.extern.slf4j.Slf4j;
import reactor.core.publisher.Mono;
```

```

@Controller
@Slf4j
public class AlertController {

    @MessageMapping("alert")
    public Mono<Void> setAlert(Mono<Alert> alertMono) {
        return alertMono
            .doOnNext(alert ->
                log.info("{} alert ordered by {} at {}",
                    alert.getLevel(),
                    alert.getOrderedBy(),
                    alert.getOrderedAt())
            )
            .thenEmpty(Mono.empty());
    }
}

```

Метод `setAlert()` обрабатывает сообщения `Alert` с маршрутом `"alert"`. Для простоты (хотя в реальной боевой обстановке это бессмысленно) он просто фиксирует тревоги в журнале. Но самое важное, что он возвращает `Mono<Void>`, указывая на отсутствие ответа и, следовательно, поддержку модели «запустил и забыл».

Код на стороне клиента не сильно отличается от моделей «запрос–ответ» или «запрос–поток»:

```

RSocketRequester tcp = requesterBuilder.tcp("localhost", 7000);
tcp
    .route("alert")
    .data(new Alert(
        Alert.Level.RED, "Craig", Instant.now()))
    .send()
    .subscribe();
log.info("Alert sent");

```

Обратите внимание, однако, что вместо `retrieveMono()` или `retrieveFlux()` клиент просто вызывает `send()`, не ожидая ответа.

Теперь посмотрим, как работает модель «канал», в которой сервер и клиент обмениваются потоками сообщений.

14.2.4 Двунаправленная передача сообщений

Во всех коммуникационных моделях, представленных до сих пор, клиент отправляет один запрос, а сервер отвечает одним или несколькими ответами или не отвечает совсем. В модели «запрос–поток» сервер может послать клиенту несколько ответов, однако клиент по-прежнему может отправить только один запрос. Но почему только сервер может получать удовольствие? Почему клиент не может отправить несколько запросов?

А вот и может, используя модель «канал»! В коммуникационной модели «канал» клиент может передать серверу несколько запросов, который, в свою очередь, может отправить несколько ответов. В ре-

зультате получается двунаправленный диалог двух сторон. Это самая гибкая из коммуникационных моделей, поддерживаемая протоколом RSocket, хотя и самая сложная.

Для демонстрации работы модели «канал» создадим службу, вычисляющую сумму вознаграждения в зависимости от суммы в счете, которая получает поток запросов и отвечает потоком ответов. Для начала определим объекты, представляющие запрос и ответ. Класс `GratuityIn`, показанный в листинге 14.8, представляет запрос, отправляемый клиентом и получаемый сервером.

Листинг 14.8 Класс, представляющий запрос для вычисления вознаграждения

```
package rsocket;

import java.math.BigDecimal;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class GratuityIn {

    private BigDecimal billTotal;
    private int percent;

}
```

Экземпляр `GratuityIn` несет два важных элемента информации, необходимых для расчета вознаграждения: общая сумма счета и процент. Класс `GratuityOut`, показанный в листинге 14.9, представляет ответ, повторяющий значения, указанные в `GratuityIn`, а также свойство `gratuity` с вычисленной суммой вознаграждения.

Листинг 14.9 Класс, представляющий ответ с вычисленной суммой вознаграждения

```
package rsocket;

import java.math.BigDecimal;

import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class GratuityOut {

    private BigDecimal billTotal;
    private int percent;

}
```

```
    private BigDecimal gratuity;  
}
```

Контроллер `GratuityController` в листинге 14.10 обрабатывает запросы на вычисление вознаграждения и очень похож на другие контроллеры, написанные нами выше в этой главе.

Листинг 14.10 Контроллер RSocket, реализующий модель «канал» и обрабатывающий поток запросов

```
package rsocket;  
  
import java.math.BigDecimal;  
import org.springframework.messaging.handler.annotation.MessageMapping;  
import org.springframework.stereotype.Controller;  
  
import lombok.extern.slf4j.Slf4j;  
import reactor.core.publisher.Flux;  
  
@Controller  
@Slf4j  
public class GratuityController {  
  
    @MessageMapping("gratuity")  
    public Flux<GratuityOut> calculate(Flux<GratuityIn> gratuityInFlux) {  
        return gratuityInFlux  
            .doOnNext(in -> log.info("Calculating gratuity: {}", in))  
            .map(in -> {  
                double percentAsDecimal = in.getPercent() / 100.0;  
                BigDecimal gratuity = in.getBillTotal()  
                    .multiply(BigDecimal.valueOf(percentAsDecimal));  
                return new GratuityOut(  
                    in.getBillTotal(), in.getPercent(), gratuity);  
            });  
    }  
}
```

Между этим и предыдущими примерами есть, однако, одно существенное отличие: этот контроллер не только возвращает, но и принимает поток `Flux`. Так же как в случае с моделью «запрос–поток», возвращаемый поток `Flux` позволяет контроллеру передать клиенту несколько значений клиенту. Но именно параметр типа `Flux` отличает модель «канал» от модели «запрос–поток». Входной параметр типа `Flux` позволяет контроллеру обрабатывать поток запросов от клиента, поступающих в метод обработчика.

Реализация модели «канал» на стороне клиента отличается от реализации модели «запрос–поток» только тем, что она посыпает серверу `Flux<GratuityIn>` вместо `Mono<GratuityIn>`, как показано в листинге 14.11.

Листинг 14.11 Клиент, посылающий и принимающий множество сообщений через открытый канал

```
RSocketRequester tcp = requesterBuilder.tcp("localhost", 7000);

Flux<GratuityIn> gratuityInFlux =
    Flux.fromArray(new GratuityIn[] {
        new GratuityIn(BigDecimal.valueOf(35.50), 18),
        new GratuityIn(BigDecimal.valueOf(10.00), 15),
        new GratuityIn(BigDecimal.valueOf(23.25), 20),
        new GratuityIn(BigDecimal.valueOf(52.75), 18),
        new GratuityIn(BigDecimal.valueOf(80.00), 15)
    })
    .delayElements(Duration.ofSeconds(1));

tcp
    .route("gratuity")
    .data(gratuityInFlux)
    .retrieveFlux(GratuityOut.class)
    .subscribe(out ->
        log.info(out.getPercent() + "% gratuity on "
            + out.getBillTotal() + " is "
            + out.getGratitude()));

```

В этом случае поток `Flux<GratuityIn>` создается статически с использованием метода `fromArray()`, но точно так же он может быть создан на основе любого другого источника данных, даже репозитория реактивных данных.

Возможно, вы заметили, что поддерживаемая коммуникационная модель RSocket на стороне сервера определяется реактивными типами, которые принимают и возвращают методы контроллера. Эта закономерность определена в табл. 14.1.

Таблица 14.1 Используемая коммуникационная модель RSocket определяется типами входных параметров и возвращаемых значений методов контроллеров

Модель RSocket	Тип входного параметра	Тип возвращаемого значения
Запрос–ответ	Mono	Mono
Запрос–поток	Mono	Flux
Запустил и забыл	Mono	Mono<Void>
Канал	Flux	Flux

У вас наверняка возник вопрос: может ли сервер принять `Flux` и вернуть `Mono`. К сожалению, такая возможность не предусмотрена. Конечно, можно представить себе обработку нескольких запросов из входящего потока `Flux` с возвратом ответа `Mono<Void>` – этакое странное сочетание моделей «канал» и «запустил и забыл», однако в RSocket не существует модели, соответствующей этому сценарию.

14.3 Передача по протоколу RSocket через WebSocket

По умолчанию взаимодействия по протоколу RSocket осуществляются через сокет TCP. Но в некоторых случаях этот вариант не подходит. Рассмотрим две ситуации:

- клиент написан на JavaScript и выполняется в веб-браузере пользователя;
- клиент должен пересечь границу шлюза или брандмауэра, чтобы получить доступ к серверу, а брандмауэр запрещает связь через произвольные порты.

Кроме того, в WebSocket отсутствует какая-либо поддержка маршрутизации, поэтому маршруты требуется определять на уровне приложения. Организуя передачу по протоколу RSocket поверх WebSocket, можно получить выгоду от встроенной поддержки маршрутизации в RSocket.

В ситуациях, подобных перечисленным выше, можно передавать данные по протоколу RSocket через WebSocket. Взаимодействие через WebSocket осуществляется по протоколу HTTP, который является основным протоколом взаимодействий для всех веб-браузеров и обычно пропускается брандмауэрами.

Чтобы получить возможность на транспортном уровне использовать WebSocket вместо TCP, нужно внести лишь несколько незначительных изменений в сервер и клиент. Прежде всего, поскольку транспортный протокол WebSocket действует поверх HTTP, нужно обеспечить поддержку обработки HTTP-запросов в приложении на стороне сервера. Проще говоря, нужно добавить следующую начальную зависимость WebFlux в спецификацию сборки проекта (если ее еще нет):

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-webflux</artifactId>
</dependency>
```

Также в конфигурации на стороне сервера нужно указать, что приложение будет использовать транспорт WebSocket, установив свойство `spring.rsocket.server.transport`. Кроме того, необходимо определить HTTP-путь для взаимодействий через RSocket, установив свойство `spring.rsocket.server.mapping-path`. Вот как могла бы выглядеть конфигурация сервера в `application.yml`:

```
spring:
  rsocket:
    server:
      transport: websocket
      mapping-path: /rsocket
```

В отличие от транспорта TCP, обменивающегося данными через определенный порт, транспорт WebSocket использует определенный путь HTTP. Поэтому нет необходимости устанавливать свойство `spring.rsocket.server.port`, как в случае с RSocket через TCP.

Это все, что нужно сделать на стороне сервера, чтобы включить транспорт WebSocket для RSocket. В остальном все будет работать точно так же, как и прежде.

На стороне клиента нужно только одно небольшое изменение. Вместо объекта, посылающего запрос через TCP, нужно создать объект, использующий WebSocket, вызвав метод `RSocketRequester.Builder.eg.websocket()`:

```
RSocketRequester requester = requesterBuilder.websocket(  
    URI.create("ws://localhost:8080/rsocket"));  
  
requester  
    .route("greeting")  
    .data("Hello RSocket!")  
    .retrieveMono(String.class)  
    .subscribe(response -> log.info("Got a response: {}", response));
```

И это все, что нужно для передачи по протоколу RSocket через WebSocket!

Итоги

- RSocket – это асинхронный двоичный протокол, предлагающий четыре коммуникационные модели: «запрос–ответ», «запрос–поток», «запустил и забыл» и «канал».
- В Spring поддержка RSocket на сервере организована в форме контроллеров и методов обработчиков с аннотацией `@MessageHandler`.
- На стороне клиента поддержка RSocket обеспечивается посредством `RSocketRequester`.
- В обоих случаях поддержка RSocket в Spring основана на реактивных типах `Flux` и `Mono` из библиотеки Reactor.
- Взаимодействия по протоколу RSocket по умолчанию осуществляются через TCP, но также есть возможность использовать WebSocket для обхода ограничений брандмауэра и клиентов, действующих в браузере.

Часть IV

Развертывание *Spring*

В этой части книги мы подготовим приложение к выпуску и посмотрим, что нужно сделать, чтобы его развернуть. Глава 15 познакомит вас с Spring Boot Actuator – расширением Spring Boot, – которое открывает возможность мониторинга действующего приложения Spring, добавляя конечные точки REST и JMX MBean. В главе 16 вы увидите, как использовать Spring Boot Admin для развертывания поверх Actuator удобного браузерного приложения администрирования, а также узнаете, как зарегистрировать клиентские приложения и защитить сервер администрирования. В главе 17 обсуждается организация доступа к bean-компонентам Spring посредством JMX MBean. Наконец, в главе 18 вы познакомитесь с процедурой развертывания приложения Spring в различных производственных окружениях, включая контейнерные, действующие под управлением Kubernetes.

Spring Boot Actuator

В этой главе рассматриваются следующие темы:

- добавление поддержки Actuator в проекты Spring Boot;
- конечные точки Actuator;
- настройка Actuator;
- защита доступа к конечным точкам Actuator.

Представьте, что вам подарили запакованный подарок и вы пытаетесь угадать, что находится внутри. Вы встряхиваете его, прикидываете размеры и вес. Возможно даже, что вы четко представляете, что там внутри. Но пока не откроете коробку, не узнаете.

Действующее приложение похоже на запакованный подарок. Вы можете обратиться к нему и сделать разумные предположения о про-исходящем внутри. Но это будут лишь предположения. Если бы был какой-то способ, с помощью которого можно заглянуть внутрь дей-ствующего приложения, посмотреть на его поведение, проверить ра-ботоспособность и, возможно, даже выполнить какие-то операции, влияющие на его работу!

В этой главе мы познакомимся с проектом Spring Boot Actuator. Он предлагает готовые к использованию средства мониторинга и оцен-ки параметров работы приложений Spring Boot. Информация, соби-раемая механизмами Actuator, доступна через набор конечных точек HTTP, а также JMX MBean. В этой главе основное внимание мы уделим конечным точкам HTTP, а конечные точки JMX рассмотрим в главе 18.

15.1 Введение в Actuator

Актуатором¹ в механике называется устройство, отвечающее за управление и перемещение основного механизма. Spring Boot Actuator в приложениях Spring Boot играет ту же роль, позволяя заглянуть внутрь действующего приложения и до определенной степени управлять его поведением.

Используя конечные точки, предоставляемые Spring Boot Actuator, можно получить ответы на вопросы о внутреннем состоянии приложения Spring Boot, например:

- Какие конфигурационные свойства доступны в приложении?
- Какие уровни журналирования используют те или иные пакеты в приложении?
- Сколько памяти потребляет приложение?
- Сколько раз была запрошена данная конечная точка HTTP?
- В каком состоянии находятся приложение и внешние службы, с которыми оно взаимодействует?

Чтобы добавить Actuator в приложение Spring Boot, достаточно добавить начальную зависимость Actuator в спецификацию сборки, например в файл Maven *pom.xml*:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

После этого в приложение будет добавлено несколько готовых конечных точек Actuator, в том числе описанные в табл. 15.1.

Таблица 15.1. Конечные точки Actuator для получения информации о действующем приложении Spring Boot и управления им

Метод HTTP	Путь	Описание
GET	/auditevents	Генерирует отчет обо всех событиях аудита, имевших место
GET	/beans	Возвращает описания всех bean-компонентов в контексте приложения Spring
GET	/conditions	Генерирует отчет о действиях механизма автоконфигурации, которые были выполнены успешно или потерпели неудачу, которые привели к созданию bean-компонентов в контексте приложения
GET	/configprops	Генерирует описание всех конфигурационных свойств с их текущими значениями
GET, POST, DELETE	/env	Генерирует отчет с перечислением всех источников свойств и свойств, полученных из них
GET	/env/{toMatch}	Возвращает описание значения одного заданного свойства из окружения
GET	/health	Возвращает агрегатную оценку работоспособности приложения и (если есть возможность) оценку работоспособности внешних зависимостей приложения

¹ Actuator – исполнительный механизм, рычаг. – Прим. перев.

Таблица 15.1 (окончание)

Метод HTTP	Путь	Описание
GET	/heapdump	Возвращает дамп динамической памяти (кучи)
GET	/httptrace	Возвращает трассировку 100 последних запросов
GET	/info	Возвращает информацию о приложении, предусмотренную разработчиком
GET	/loggers	Возвращает список пакетов в приложении с настроенными и фактическими уровнями журналирования
GET, POST	/loggers/{name}	Возвращает настроенный и фактический уровень журналирования для указанного пакета; фактический уровень журналирования можно изменить с помощью запроса POST
GET	/mappings	Генерирует отчет обо всех маршрутах HTTP и соответствующих им методах обработчиков
GET	/metrics	Возвращает список всех категорий метрик
GET	/metrics/{name}	Возвращает многомерное множество значений указанной метрики
GET	/scheduledtasks	Возвращает список всех запланированных заданий
GET	/threaddump	Генерирует отчет со списком всех потоков выполнения в приложении

В дополнение к конечным точкам HTTP, перечисленным в табл. 15.1, кроме /heapdump, представляются также аналогичные конечные точки JMX MBean (мы рассмотрим их в главе 17).

15.1.1 Настройка базового пути Actuator

По умолчанию пути ко всем конечным точкам, перечисленным в табл. 15.1, начинаются с префикса /actuator. То есть чтобы получить информацию о работоспособности вашего приложения, например, нужно выполнить запрос GET к конечной точке /actuator/health.

Префикс можно изменить, определив свойство management.endpoint.web.base-path. Например, чтобы задать префикс /management, установите свойство management.endpoints.web.base-path, как показано ниже:

```
management:
  endpoints:
    web:
      base-path: /management
```

После этого информацию о работоспособности приложения можно будет получить, выполнив запрос GET к конечной точке /management/health.

Независимо от выбора базового пути, все конечные точки в этой главе для краткости будут указываться без него. Например, упоминая конечную точку /health, я буду подразумевать конечную точку /{базовый путь}/health, или /actuator/health, если базовый путь не был изменен.

15.1.2 Включение и отключение конечных точек Actuator

На первых порах можно заметить, что по умолчанию включена только конечная точка /health. Большинство конечных точек Actuator воз-

вращают конфиденциальную информацию и требуют защиты. Вы можете использовать Spring Security для блокировки Actuator, но поскольку Actuator не защищен сам по себе, большинство конечных точек по умолчанию отключены, поэтому вам придется выбрать, какие конечные точки включить.

Для управления доступными конечными точками можно использовать два конфигурационных свойства: `management.endpoints.web.exposure.include` и `management.endpoints.web.exposure.exclude`. В `management.endpoints.web.exposure.include` нужно указать, какие конечные точки должны быть включены. Например, вот как можно открыть доступ только к конечным точкам `/health`, `/info`, `/beans` и `/conditions`:

```
management:  
  endpoints:  
    web:  
      exposure:  
        include: health,info,beans,conditions
```

В свойстве `management.endpoints.web.exposure.include` также можно использовать подстановочный символ звездочка (*), чтобы указать, что все конечные точки Actuator должны быть включены:

```
management:  
  endpoints:  
    web:  
      exposure:  
        include: '*'
```

Чтобы включить все конечные точки, кроме нескольких, обычно проще включить их все с помощью звездочки, а затем явно исключить нежелательные. Например, чтобы предоставить доступ ко всем конечным точкам, кроме `/threaddump` и `/heapdump`, можно установить свойства `management.endpoints.web.exposure.include` и `management.endpoints.web.exposure.exclude`, как показано ниже:

```
management:  
  endpoints:  
    web:  
      exposure:  
        include: '*'  
        exclude: threaddump,heapdump
```

Если вы решите включить другие конечные точки, помимо `/health` и `/info`, то было бы неплохо настроить Spring Security, чтобы ограничить доступ к другим конечным точкам. О защите конечных точек Actuator мы поговорим в разделе 15.4, а пока давайте посмотрим, как использовать конечные точки HTTP, предоставляемые Actuator.

15.2 Использование конечных точек Actuator

Actuator может подарить нам настоящую сокровищницу с интересной и полезной информацией о действующем приложении, доступную через конечные точки HTTP, перечисленные в табл. 15.1. Эти конечные точки можно использовать как любой REST API с помощью любого HTTP-клиента, включая `RestTemplate` и `WebClient` в Spring, из приложений на JavaScript или даже с помощью клиента командной строки `curl`.

Для изучения конечных точек Actuator в этой главе мы используем клиента командной строки `curl`. В главе 16 я познакомлю вас с проектом Spring Boot Admin, предлагающим удобное веб-приложение для доступа к конечным точкам Actuator.

Чтобы получить представление о том, какие конечные точки доступны, можно выполнить запрос GET к базовому пути Actuator и получить в ответ ссылки HATEOAS для всех конечных точек. При использовании `curl` можно получить примерно такой ответ (здесь приводится только часть ответа для экономии места):

```
$ curl localhost:8080/actuator
{
  "_links": {
    "self": {
      "href": "http://localhost:8080/actuator",
      "templated": false
    },
    "auditevents": {
      "href": "http://localhost:8080/actuator/auditevents",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8080/actuator/beans",
      "templated": false
    },
    "health": {
      "href": "http://localhost:8080/actuator/health",
      "templated": false
    },
    ...
  }
}
```

Поскольку разные библиотеки могут создавать дополнительные конечные точки Actuator, а некоторые конечные точки могут быть выключены, то фактические результаты могут различаться в разных приложениях.

В любом случае набор ссылок, возвращаемых при обращении к базовому пути Actuator, может служить перечнем всего, что готов предложить Actuator. Давайте начнем наше исследование ландшафта Ac-

tuator с двух конечных точек, возвращающих важную информацию о приложении: `/health` и `/info`.

15.2.1 Получение важной информации о приложении

Когда мы обращаемся к врачу, то обычно слышим два основных вопроса: как вас зовут и как вы себя чувствуете? Разные врачи и медсестры могут по-разному формулировать эти вопросы, но в любом случае они хотят немного узнать о человеке и о причинах обращения к ним.

На эти же важные вопросы отвечают конечные точки `/info` и `/health`. Конечная точка `/info` сообщает краткую информацию о приложении, а конечная точка `/health` – о его работоспособности.

ЗАПРОС ИНФОРМАЦИИ О ПРИЛОЖЕНИИ

Чтобы получить немного информации о действующем приложении Spring Boot, можно обратиться к конечной точке `/info`. Однако по умолчанию она не очень информативна. Вот, например, что можно получить, выполнив запрос с помощью `curl`:

```
$ curl localhost:8080/actuator/info
{}
```

Получив такой вывод, мы могли бы подумать, что конечная точка `/info` совершенно бесполезна, но это не так. На самом деле эту точку можно представить как чистый холст, на котором мы должны изобразить все, что может понадобиться.

Сделать это можно несколькими способами, но самый простой – создать одно или несколько свойств конфигурации с именами, начинающимися с префикса `info`. Например, предположим, что мы решили экспорттировать через конечную точку `/info` контактную информацию службы поддержки, включая адрес электронной почты и номер телефона. Для этого можно определить в файле `application.yml` следующие свойства:

```
info:
  contact:
    email: support@tacocloud.com
    phone: 822-625-6831
```

Ни свойство `info.contact.email`, ни свойство `info.contact.phone` не имеют особого значения для Spring Boot и любых компонентов, находящихся в контексте приложения. Однако в силу наличия префикса `info` конечная точка `/info` теперь будет отображать эти свойства в своем ответе:

```
{
  "contact": {
    "email": "support@tacocloud.com",
```

```

        "phone": "822-625-6831"
    }
}

```

В разделе 15.3.1 мы рассмотрим еще несколько способов экспортации полезной информации через конечную точку `/info`.

ПРОВЕРКА СОСТОЯНИЯ ПРИЛОЖЕНИЯ

В ответ на HTTP-запрос GET конечная точка `/health` вернет простой текст в формате JSON, описывающий состояние работоспособности приложения. Например, вот что можно увидеть, выполнив такой запрос с помощью `curl`:

```
$ curl localhost:8080/actuator/health
{"status":"UP"}
```

У кого-то из вас может возникнуть вопрос: в чем же полезность информации, сообщающей, что приложение работает (`UP`)? И что сообщит эта конечная точка, если приложение не работает?

Как оказывается, показанный здесь статус является совокупным индикатором, объединяющим несколько других индикаторов работоспособности, сообщающих о работоспособности внешних систем, с которыми взаимодействует приложение, таких как базы данных, брокеров сообщений и даже компонентов Spring Cloud, таких как Eureka и Config Server. Каждый индикатор работоспособности может находиться в одном из следующих состояний:

- `UP` – внешняя система работает и доступна;
- `DOWN` – внешняя система не работает или недоступна;
- `UNKNOWN` – состояние внешней системы не ясно;
- `OUT_OF_SERVICE` – внешняя система доступна, но в данный момент не может обслуживать приложение.

Состояния всех индикаторов работоспособности объединяются в общий индикатор в соответствии со следующими правилами:

- если все индикаторы работоспособности находятся в состоянии `UP`, то совокупный индикатор получает состояние `UP`;
- если один или несколько индикаторов работоспособности находятся в состоянии `DOWN`, то совокупный индикатор получает состояние `DOWN`;
- если один или несколько индикаторов работоспособности находятся в состоянии `OUT_OF_SERVICE`, то совокупный индикатор получает состояние `OUT_OF_SERVICE`;
- состояния `UNKNOWN` игнорируются и не влияют на выбор значения для совокупного индикатора.

По умолчанию в ответ на запрос конечная точка `/health` возвращает только совокупный статус. Однако есть возможность настроить свойство `management.endpoint.health.show-details` так, чтобы конечная

точка возвращала полную информацию обо всех индикаторах работоспособности:

```
management:
  endpoint:
    health:
      show-details: always
```

Свойство `management.endpoint.health.show-details` по умолчанию имеет значение `never`, но ему можно присвоить значение `always`, чтобы всегда возвращалась полная информация обо всех индикаторах работоспособности, или значение `when-authorized`, чтобы полная информация отображалась только авторизованным клиентам.

Если теперь отправить запрос GET конечной точке `/health`, она вернет полную информацию об индикаторах работоспособности. Вот как может выглядеть такая информация для службы, интегрированной с документной базой данных Mongo:

```
{
  "status": "UP",
  "details": {
    "mongo": {
      "status": "UP",
      "details": {
        "version": "3.5.5"
      }
    },
    "diskSpace": {
      "status": "UP",
      "details": {
        "total": 499963170816,
        "free": 177284784128,
        "threshold": 10485760
      }
    }
  }
}
```

Все приложения с любыми внешними зависимостями будут включать индикатор работоспособности файловой системы с именем `diskSpace`. Он сообщает о работоспособности файловой системы (надеюсь, UP) и объеме оставшегося свободного пространства. Если доступное дисковое пространство окажется ниже порогового значения, то индикатор перейдет в состояние DOWN.

В предыдущем примере также присутствует индикатор работоспособности `mongo`, сообщающий о состоянии базы данных Mongo, в том числе и ее версию. Механизм автоконфигурации гарантирует, что в ответе от конечной точки `/health` появятся только индикаторы, относящиеся к приложению. Помимо индикаторов работоспособности `mongo` и `diskSpace` фреймворк Spring Boot предоставляет также инди-

каторы работоспособности для некоторых других внешних баз данных и систем, включая:

- Cassandra;
- Config Server;
- Couchbase;
- Eureka;
- Hystrix;
- источники данных JDBC;
- Elasticsearch;
- InfluxDB;
- брокеры сообщений JMS;
- LDAP;
- серверы электронной почты;
- Neo4j;
- брокеры сообщений Rabbit;
- Redis;
- Solr.

Сторонние библиотеки тоже могут предоставлять свои индикаторы работоспособности. Кроме того, в разделе 15.3.2 я покажу, как написать свой индикатор работоспособности.

Как видите, конечные точки `/health` и `/info` возвращают обобщенную информацию о действующем приложении. Другие конечные точки Actuator позволяют получить представление о конфигурации приложения. Давайте посмотрим, как Actuator может экспортить конфигурацию приложения.

15.2.2 Просмотр сведений о конфигурации

Помимо получения общей информации о приложении часто полезно знать, как оно настроено. Какие компоненты имеются в контексте приложения? Какие действия выполнены или не выполнены механизмом автоконфигурации? Какие свойства доступны приложению? Каким контроллерам передаются те или иные HTTP-запросы для обработки? Какой уровень журналирования установлен для тех или иных пакетов или классов?

На эти вопросы отвечают конечные точки `/beans`, `/conditions`, `/env`, `/configprops`, `/mappings` и `/loggers`. Некоторые конечные точки, такие как `/env` и `/loggers`, позволяют даже изменять конфигурацию запущенного приложения на лету. Давайте посмотрим, как каждая из этих конечных точек сообщает информацию о конфигурации приложения, начав с конечной точки `/beans`.

ПОЛУЧЕНИЕ ОТЧЕТА О КОМПОНЕНТАХ

Наиболее важной конечной точкой для изучения контекста приложения Spring является конечная точка `/beans`. Она возвращает отчет в формате JSON, описывающий каждый отдельный компонент в кон-

тексте приложения, его тип и любые другие компоненты, внедренные в него.

Полный ответ на запрос GET к конечной `/beans` мог бы заполнить всю эту главу. Поэтому мы рассмотрим только небольшой фрагмент, включающий одну запись, соответствующую одному компоненту:

```
{
  "contexts": {
    "application-1": {
      "beans": {
        ...
        "ingredientsController": {
          "aliases": [],
          "scope": "singleton",
          "type": "tacos.ingredients.IngredientsController",
          "resource": "file [/Users/habuma/Documents/Workspaces/
            TacoCloud/ingredient-service/target/classes/tacos/
            ingredients/IngredientsController.class]",
          "dependencies": [
            "ingredientRepository"
          ],
        },
        ...
      },
      "parentId": null
    }
  }
}
```

Корневым элементом в ответе является `contexts`. Он включает по одному подэлементу для каждого контекста приложения Spring. В каждом контексте приложения есть элемент `beans` со сведениями обо всех компонентах в контексте приложения.

В предыдущем примере показан компонент `ingredientsController`. Как видите, у него нет псевдонимов, он имеет область действия как синглтон и тип `tacos.ingredients.IngredientsController`. Кроме того, свойство `resource` сообщает путь к файлу класса, где определен компонент. А свойство `dependencies` перечисляет все другие компоненты, которые внедрены в данный компонент. В этом случае в компонент `ingredientsController` внедрен компонент с именем `ingredientRepository`.

ИССЛЕДОВАНИЕ АВТОКОНФИГУРАЦИИ

Как рассказывалось выше в этой книге, автоконфигурация – одна из самых мощных особенностей Spring Boot. Однако иногда может возникнуть вопрос, почему что-то было сконфигурировано автоматически. Или, наоборот, что-то должно было быть сконфигурировано автоматически, но этого не произошло. В таких случаях можно выполнить запрос GET к конечной точке `/conditions` и получить объяснение, что произошло в автоконфигурации.

Отчет механизма автоконфигурации, возвращаемый конечной точкой `/conditions`, разделен на три части: положительные случаи (условная автоконфигурация была успешно выполнена), отрицательные случаи (условная автоконфигурация потерпела неудачу) и безусловные классы. В следующем фрагменте ответа на запрос к `/conditions` показан пример каждой части:

```
{
  "contexts": {
    "application-1": {
      "positiveMatches": {
        ...
        "MongoDataAutoConfiguration#mongoTemplate": [
          {
            "condition": "OnBeanCondition",
            "message": "@ConditionalOnMissingBean (types: org.springframework.data.mongodb.core.MongoTemplate; SearchStrategy: all) did not find any beans"
          }
        ],
        ...
      },
      "negativeMatches": {
        ...
        "DispatcherServletAutoConfiguration": {
          "notMatched": [
            {
              "condition": "OnClassCondition",
              "message": "@ConditionalOnClass did not find required class 'org.springframework.web.servlet.DispatcherServlet'"
            }
          ],
          "matched": []
        },
        ...
      },
      "unconditionalClasses": [
        ...
        "org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration",
        ...
        ]
      }
    }
  }
}
```

В разделе `PositiveMatches` можно видеть, что компонент `MongoTemplate` был настроен с помощью механизма автоконфигурации, потому что его еще не существовало. Это действие механизма автоконфигурации может быть вызвано аннотацией `@ConditionalOnMissingBean`,

которая требует выполнить настройку компонента, если он не был настроен явно. Но в этом случае компоненты MongoTemplate просто не были найдены в контексте приложения, поэтому механизм автоконфигурации выполнил настройку.

В разделе `negativeMatches` можно видеть, что механизм автоконфигурации Spring Boot заметил необходимость настройки `DispatcherServlet`. Но условная аннотация `@ConditionalOnClass` потерпела неудачу из-за отсутствия `DispatcherServlet`.

Наконец, компонент `ConfigurationPropertiesAutoConfiguration` был настроен без всяких условий, как показано в разделе `unconditionalClasses`. В основе работы Spring Boot лежат конфигурационные свойства, поэтому любые свойства, относящиеся к конфигурации, должны настраиваться автоматически и без всяких условий.

ПРОВЕРКА СВОЙСТВ ОКРУЖЕНИЯ И КОНФИГУРАЦИИ

В дополнение к информации о взаимосвязях компонентов приложения нередко также бывает полезно знать, какие свойства окружения доступны и какие конфигурационные свойства были внедрены в компоненты.

Послав запрос `GET` конечной точке `/env`, вы получите длинный список, включающий свойства из всех источников, доступных приложению Spring. Сюда входят и переменные окружения, и системные свойства JVM, и файлы `application.properties` и `application.yml`, и даже Spring Cloud Config Server (если приложение является клиентом Config Server).

В листинге 15.1 показан сильно сокращенный список, который можно получить от конечной точки `/env`, чтобы вы могли получить некоторое представление о том, какую информацию она предоставляет.

Листинг 15.1 Результат обращения к конечной точке `/env`

```
$ curl localhost:8080/actuator/env
{
  "activeProfiles": [
    "development"
  ],
  "propertySources": [
    ...
    {
      "name": "systemEnvironment",
      "properties": {
        "PATH": {
          "value": "/usr/bin:/bin:/usr/sbin:/sbin",
          "origin": "System Environment Property \"PATH\""
        },
        ...
        "HOME": {
          "value": "/Users/habuma",
          "origin": "System Environment Property \"HOME\""
        }
      }
    }
  ]
}
```

```

        }
    },
},
{
  "name": "applicationConfig: [classpath:/application.yml]",
  "properties": {
    "spring.application.name": {
      "value": "ingredient-service",
      "origin": "class path resource [application.yml]:3:11"
    },
    "server.port": {
      "value": 8080,
      "origin": "class path resource [application.yml]:9:9"
    },
    ...
  }
},
...
]
}

```

Полный ответ конечной точки `/env` содержит намного больше информации, однако и в листинге 15.1 имеются некоторые заслуживающие внимания сведения. Во-первых, обратите внимание на поле с именем `activeProfiles` в верхней части ответа. Оно указывает, что в данном случае активен профиль `development`. Если будут активны какие-то другие профили, они также будут перечислены здесь.

Следующее интересное поле: `propertySources` – массив источников свойств в окружении приложения. В листинге 15.1 показаны только два источника свойств: `systemEnvironment` и `applicationConfig`, ссылающийся на файл `application.yml`.

Для каждого источника приводится список всех свойств, предоставляемых этим источником, вместе с их значениями. В случае источника `application.yml` поле `origin` в каждом свойстве точно указывает, где установлено свойство, включая строку и столбец в `application.yml`.

Конечная точка `/env` также может вернуть конкретное свойство, если его имя указать во втором элементе пути. Например, чтобы получить информацию о свойстве `server.port`, нужно отправить запрос GET с путем `/env/server.port`, например:

```
$ curl localhost:8080/actuator/env/server.port
{
  "property": {
    "source": "systemEnvironment", "value": "8080"
  },
  "activeProfiles": [ "development" ],
  "propertySources": [
    { "name": "server.ports" },
    { "name": "mongo.ports" },
  ]
}
```

```
{
  "name": "systemProperties" },
  { "name": "systemEnvironment",
    "property": {
      "value": "8080",
      "origin": "System Environment Property \\"SERVER_PORT\\"
    }
  },
  { "name": "random" },
  { "name": "applicationConfig: [classpath:/application.yml]",
    "property": {
      "value": 0,
      "origin": "class path resource [application.yml]:9:9"
    }
  },
  { "name": "springCloudClientHostInfo" },
  { "name": "refresh" },
  { "name": "defaultProperties" },
  { "name": "Management Server" }
]
}
```

Как видите, в ответе все еще представлены все источники свойств, но какая-либо дополнительная информация приводится только для тех из них, которые определяют указанное свойство. В этом случае оба источника, `systemEnvironment` и `application.yml`, определяют значение свойства `server.port`. Поскольку `systemEnvironment` имеет приоритет над любым из перечисленных ниже источников, свойство `server.port` получает значение 8080 из него. Окончательное значение отображается вверху в поле `property`.

Конечная точка `/env` может использоваться не только для чтения значений свойств. Отправив запрос POST в конечную точку `/env` с документом JSON с полями `name` и `value`, также можно менять значения свойств в действующем приложении. Например, чтобы присвоить свойству `tacocloud.discount.code` значение `TACOS1234`, можно с помощью `curl` отправить запрос POST, как показано ниже:

```
$ curl localhost:8080/actuator/env \
  -d'{"name":"tacocloud.discount.code","value":"TACOS1234"}' \
  -H "Content-type: application/json"
{"tacocloud.discount.code":"TACOS1234"}
```

После отправки этого запроса его обновленное значение возвращается в ответе. Позже, решив, что это свойство больше не нужно, вы можете отправить запрос `DELETE` конечной точке `/env`, чтобы удалить все свойства, созданные через эту конечную точку:

```
$ curl localhost:8080/actuator/env -X DELETE
{"tacocloud.discount.code":"TACOS1234"}
```

Какой бы полезной ни была установка свойств через Actuator API, важно понимать, что любые свойства, установленные с помощью

POST-запроса к конечной точке `/env`, применяются только к экземпляру приложения, получившему запрос, являются временными и будут потеряны после перезапуска приложения.

ИССЛЕДОВАНИЕ МАРШРУТИЗАЦИИ ЗАПРОСОВ

Модель программирования Spring MVC (и Spring WebFlux) упрощает обработку HTTP-запросов, позволяя настраивать маршруты с помощью аннотаций, но иногда бывает сложно получить общее представление обо всех типах HTTP-запросов, которые может обрабатывать приложение, и о компонентах, их обрабатывающих.

По этой причине Actuator добавляет конечную точку `/mappings`, возвращающую единый отчет с описанием всех обработчиков HTTP-запросов, будь то контроллеры Spring MVC или конечные точки Actuator. Чтобы получить список всех конечных точек в приложении Spring Boot, можно выполнить запрос GET к конечной точке `/mappings`, как показано в листинге 15.2.

Листинг 15.2 Маршруты обработки HTTP-запросов

```
$ curl localhost:8080/actuator/mappings | jq
{
  "contexts": {
    "application-1": {
      "mappings": {
        "dispatcherHandlers": {
          "webHandler": [
            ...
            {
              "predicate": "[/{[/ingredients}],methods=[GET]]",
              "handler": "public
reactor.core.publisher.Flux<tacos.ingredients.Ingredient>
tacos.ingredients.IngredientsController.allIngredients()",
              "details": {
                "handlerMethod": {
                  "className": "tacos.ingredients.IngredientsController",
                  "name": "allIngredients",
                  "descriptor": "()Lreactor/core/publisher/Flux;"
                },
                "handlerFunction": null,
                "requestMappingConditions": {
                  "consumes": [],
                  "headers": [],
                  "methods": [
                    "GET"
                  ],
                  "params": [],
                  "patterns": [
                    "/ingredients"
                  ],
                  "produces": []
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

```

        }
    },
},
...,
]
},
},
"parentId": "application-1"
},
"bootstrap": {
    "mappings": {
        "dispatcherHandlers": {}
    },
    "parentId": null
}
}
}
}

```

Здесь ответ, полученный командой `curl`, передается утилите `jq` (<https://stedolan.github.io/jq/>), которая, среди прочего, форматирует документ JSON в удобочитаемый вид. Я сократил этот ответ и оставил в нем только один обработчик запросов. В частности, здесь можно видеть, что запросы GET к конечной точке `/ingredients` обрабатываются методом `allIngredients()` класса `IngredientsController`.

УПРАВЛЕНИЕ УРОВНЯМИ ЖУРНАЛИРОВАНИЯ

Журналирование – важная функция любого приложения. Журналы можно использовать как средство аудита, а также для отладки.

Выбор уровня журналирования подобен поиску золотой середины. Если установить слишком подробный уровень журналирования, то в журналы будет записываться слишком много информации, и это может затруднить поиск полезных сведений. С другой стороны, если установить слишком поверхностный уровень журналирования, то в журналах может не оказаться информации, важной для понимания работы приложения.

Уровни журналирования обычно устанавливаются для каждого пакета отдельно. Если вам станет интересно, какие уровни журналирования установлены в вашем действующем приложении Spring Boot, то вы сможете отправить запрос GET конечной точке `/loggers` и получить эту информацию, как показано ниже:

```
{
    "levels": [ "OFF", "ERROR", "WARN", "INFO", "DEBUG", "TRACE" ],
    "loggers": {
        "ROOT": {
            "configuredLevel": "INFO", "effectiveLevel": "INFO"
        },
        ...
    },
    "org.springframework.web": {
        "configuredLevel": null, "effectiveLevel": "INFO"
    }
}
```

```
        },
    ...
    "tacos": {
        "configuredLevel": null, "effectiveLevel": "INFO"
    },
    "tacos.ingredients": {
        "configuredLevel": null, "effectiveLevel": "INFO"
    },
    "tacos.ingredients.IngredientServiceApplication": {
        "configuredLevel": null, "effectiveLevel": "INFO"
    }
}
```

Ответ конечной точки `/loggers` начинается со списка всех поддерживаемых уровней журналирования. Далее следует элемент `loggers` с перечнем сведений об уровнях журналирования всех пакетов в приложении. Свойство `configuredLevel` показывает явно настроенный уровень журналирования (или значение `null`, если он не был настроен явно). Свойство `effectiveLevel` сообщает фактический уровень журналирования, возможно унаследованный от родительского пакета или от корневого механизма журналирования.

В этом отрывке показаны уровни журналирования только для корневого механизма журналирования и четырех пакетов, но в реальности полный ответ будет включать сведения об уровнях журналирования для всех отдельных пакетов в приложении, в том числе для библиотек. Есть возможность конкретизировать запрос, чтобы получить сведения только для определенного пакета. Для этого нужно указать имя пакета в виде дополнительного компонента пути в запросе.

Например, чтобы узнать уровни журналирования, установленные для пакета `tacocloud.ingredients`, можно выполнить запрос с путем `/loggers/tacos.ingredients`:

```
{  
    "configuredLevel": null,  
    "effectiveLevel": "INFO"  
}
```

Конечная точка `/loggers` позволяет не только получить текущие значения уровней журналирования для пакетов приложений, но также изменить их, для чего нужно отправить ей запрос POST. Например, предположим, что мы решили установить уровень журналирования DEBUG для пакета `tacocloud.ingredients`. Вот как это можно сделать с помощью команды curl:

```
$ curl localhost:8080/actuator/loggers/tacos/ingredients \
    -d'{"configuredLevel":"DEBUG"}' \
    -H"Content-type: application/json"
```

После этого можно отправить запрос GET с путем `/loggers/tacos/ingredients`, как показано ниже, и увидеть произведенные изменения:

```
{
  "configuredLevel": "DEBUG",
  "effectiveLevel": "DEBUG"
}
```

Обратите внимание, что прежде свойство `configuredLevel` имело значение `null`, а после изменения запросом POST ему было присвоено значение `DEBUG`. Это изменение также затронуло свойство `effectiveLevel`. Но, что особенно важно, если какой-либо код в этом пакете осуществляет журналирование какой-либо информации на уровне `DEBUG`, то она будет записываться в файлы журналов.

15.2.3 Наблюдение за действиями приложения

Иногда бывает полезно понаблюдать за действиями, выполняемыми работающим приложением: какие виды HTTP-запросов обрабатываются или сколько потоков выполнения задействовано. Для этой цели Actuator предоставляет конечные точки `/httptrace`, `/threaddump` и `/heapdump`.

Конечная точка `/heapdump`, пожалуй, самая сложная для подробного описания. Она выгружает скатый gzip-файл с дампом кучи HPROF, который можно использовать для исследования проблем с памятью или потоками. Для полного описания конечной точки `/heapdump` потребовалось бы слишком много места, да и исследование дампа кучи – весьма сложная тема, поэтому я не буду обсуждать эту конечную точку дальше.

Наблюдение за обработкой HTTP-запросов

Конечная точка `/httptrace` сообщает детали о последних 100 запросах, обработанных приложением, включая метод и путь запроса, отметку времени, когда запрос был обработан, заголовки запроса и ответа, а также время, затраченное на обработку запроса.

В следующем фрагменте кода JSON показана одна запись из ответа, возвращаемого конечной точкой `/httptrace`:

```
{
  "traces": [
    {
      "timestamp": "2020-06-03T23:41:24.494Z",
      "principal": null,
      "session": null,
      "request": {
        "method": "GET",
        "uri": "http://localhost:8080/ingredients",
        "headers": {
          "Host": ["localhost:8080"],
          "User-Agent": ["curl/7.54.0"],
          "Accept": ["*/*"]
        },
      }
    }
  ]
}
```

```

        "remoteAddress": null
    },
    "response": {
        "status": 200,
        "headers": {
            "Content-Type": ["application/json;charset=UTF-8"]
        }
    },
    "timeTaken": 4
},
...
]
}

```

Эта информация, безусловно, очень может пригодиться для отладки, но еще интереснее накопить трассировочную информацию за продолжительный период времени и потом проанализировать нагрузку на приложение в разные периоды времени, а также вывести отношение успешно обработанных запросов к запросам, потерпевшим неудачу, по коду состояния в ответе. В главе 16 вы увидите, как Spring Boot Admin собирает эту информацию в форме графа, визуализирующего информацию трассировки HTTP за определенный период времени.

Мониторинг потоков

Помимо трассировки HTTP-запросов также бывает полезно понаблюдать за активностью потоков выполнения, чтобы определить, что происходит в приложении. Конечная точка `/threaddump` создает моментальный снимок активности текущего потока. Следующий фрагмент из ответа `/threaddump` дает представление о том, что возвращает эта конечная точка:

```

{
    "threadName": "reactor-http-nio-8",
    "threadId": 338,
    "blockedTime": -1,
    "blockedCount": 0,
    "waitedTime": -1,
    "waitedCount": 0,
    "lockName": null,
    "lockOwnerId": -1,
    "lockOwnerName": null,
    "inNative": true,
    "suspended": false,
    "threadState": "RUNNABLE",
    "stackTrace": [
        {
            "methodName": "kevent0",
            "fileName": "KQueueArrayWrapper.java",
            "lineNumber": -2,
        }
    ]
}

```

```
"className": "sun.nio.ch.KQueueArrayWrapper",
"nativeMethod": true
},
{
  "methodName": "poll",
  "fileName": "KQueueArrayWrapper.java",
  "lineNumber": 198,
  "className": "sun.nio.ch.KQueueArrayWrapper",
  "nativeMethod": false
},
...
],
"lockedMonitors": [
  {
    "className": "io.netty.channel.nio.SelectedSelectionKeySet",
    "identityHashCode": 1039768944,
    "lockedStackDepth": 3,
    "lockedStackFrame": {
      "methodName": "lockAndDoSelect",
      "fileName": "SelectorImpl.java",
      "lineNumber": 86,
      "className": "sun.nio.ch.SelectorImpl",
      "nativeMethod": false
    }
  },
  ...
],
"lockedSynchronizers": [],
"lockInfo": null
}
```

Полный отчет, возвращаемый конечной точкой `/threaddump`, включает все потоки выполнения, действующие внутри приложения. Для экономии места я показал запись лишь для одного потока. Как видите, кроме всего прочего, она включает сведения о блокировке и состоянии блокировки потока. Существует также возможность получить трассировку стека, дающую некоторое представление о том, на выполнение каких участков кода тратится больше всего времени.

Поскольку конечная точка `/threaddump` возвращает моментальный снимок активности потока, действительный только на момент запроса, то иногда бывает непросто составить полное представление о поведении потоков с течением времени. В главе 16 я покажу, как Spring Boot Admin позволяет использовать конечную точку `/threaddump` в режиме реального времени.

15.2.4 Получение метрик времени выполнения

Конечная точка `/metrics` сообщает информацию о множестве метрик, генерируемых действующим приложением, включая потребление памяти и процессора, затраты времени на сборку мусора и обработку

HTTP-запросов. По умолчанию Actuator поддерживает более двух десятков категорий метрик, о чем свидетельствует следующий список, полученный запросом GET к конечной точке `/metrics`:

```
$ curl localhost:8080/actuator/metrics | jq
{
  "names": [
    "jvm.memory.max",
    "process.files.max",
    "jvm.gc.memory.promoted",
    "http.server.requests",
    "system.load.average.1m",
    "jvm.memory.used",
    "jvm.gc.max.data.size",
    "jvm.memory.committed",
    "system.cpu.count",
    "logback.events",
    "jvm.buffer.memory.used",
    "jvm.threads.daemon",
    "system.cpu.usage",
    "jvm.gc.memory.allocated",
    "jvm.threads.live",
    "jvm.threads.peak",
    "process.uptime",
    "process.cpu.usage",
    "jvm.classes.loaded",
    "jvm.gc.pause",
    "jvm.classes.unloaded",
    "jvm.gc.live.data.size",
    "process.files.open",
    "jvm.buffer.count",
    "jvm.buffer.total.capacity",
    "process.start.time"
  ]
}
```

Список содержит так много метрик, что было бы невозможно обсудить их все в одной главе. Поэтому сосредоточимся только на категории `http.server.requests` и на ее примере посмотрим, как можно использовать конечную точку `/metrics`.

Если вместо простого запроса к `/metrics` отправить запрос GET к `/metrics/{имя метрики}`, то в ответ вернется более подробная информация о метриках из указанной категории. В случае с `http.server.requests` запрос GET к `/metrics/http.server.requests` вернет примерно такие данные:

```
$ curl localhost:8080/actuator/metrics/http.server.requests
{
  "name": "http.server.requests",
  "measurements": [
    { "statistic": "COUNT", "value": 2103 },
```

```

    { "statistic": "TOTAL_TIME", "value": 18.086334315 },
    { "statistic": "MAX", "value": 0.028926313 }
],
"availableTags": [
    { "tag": "exception",
      "values": [ "ResponseStatusException",
                  "IllegalArgumentException", "none" ] },
    { "tag": "method", "values": [ "GET" ] },
    { "tag": "uri",
      "values": [
          "/actuator/metrics/{requiredMetricName}",
          "/actuator/health", "/actuator/info", "/ingredients",
          "/actuator/metrics", "/**" ] },
    { "tag": "status", "values": [ "404", "500", "200" ] }
]
}

```

Наиболее важной частью в этом ответе является раздел `measurements`, включающий все метрики из запрошенной категории. В данном случае мы видим, что было обработано 2103 HTTP-запроса. Общее время, затраченное на обработку этих запросов, составило 18,086334315 с, а максимальное время, затраченное на обработку одного запроса, – 0,028926313 с.

Эти общие метрики достаточно интересны, но есть возможность еще больше конкретизировать запрос и получить еще более подробную информацию, используя для этого теги, перечисленные в разделе `availableTags`. Например, мы знаем, что было обработано 2103 запроса, но мы не знаем, сколько из них было обработано успешно с кодом состояния в ответе HTTP 200, а сколько завершились ошибкой с кодом HTTP 404 или HTTP 500. Используя тег `status`, можно получить метрики для всех запросов, обработка которых завершилась ответом с кодом состояния HTTP 404, например:

```

$ curl localhost:8080/actuator/metrics/http.server.requests? \
  tag=status:404
{
  "name": "http.server.requests",
  "measurements": [
    { "statistic": "COUNT", "value": 31 },
    { "statistic": "TOTAL_TIME", "value": 0.522061212 },
    { "statistic": "MAX", "value": 0 }
],
"availableTags": [
    { "tag": "exception",
      "values": [ "ResponseStatusException", "none" ] },
    { "tag": "method", "values": [ "GET" ] },
    { "tag": "uri",
      "values": [
          "/actuator/metrics/{requiredMetricName}", "/**" ] }
]
}

```

Указав имя и значение тега в атрибуте запроса `tag`, в ответ вы получите метрики, соответствующие запросам, в ответ на которые был отправлен код состояния HTTP 404. В данном случае 31 запрос привел к ошибке 404, и на их обслуживание было потрачено 0,522061212 с. Более того, здесь видно, что некоторые из неудачных запросов были запросами GET к конечной точке `/actuator/metrics/{requiredMetricsName}` (хотя здесь не видно, какие значения получал компонент путем `{requiredMetricsName}`), а другие – к каким-то другим конечным точкам, соответствующим универсальному пути `/**`.

А что, если понадобится узнать, сколько из таких неудачных запросов соответствовали пути `/**`? Для этого нужно просто указать тег `uri` в запросе, например:

```
% curl "localhost:8080/actuator/metrics/http.server.requests? \
tag=status:404&tag=uri:/**"
{
  "name": "http.server.requests",
  "measurements": [
    { "statistic": "COUNT", "value": 30 },
    { "statistic": "TOTAL_TIME", "value": 0.519791548 },
    { "statistic": "MAX", "value": 0 }
  ],
  "availableTags": [
    { "tag": "exception", "values": [ "ResponseStatusException" ] },
    { "tag": "method", "values": [ "GET" ] }
  ]
}
```

Теперь можно видеть, что было получено 30 запросов, содержащих некоторый путь, совпадающий с `/**`, в ответ на которые был отправлен код состояния HTTP 404, и общее время обработки таких запросов составило 0,519791548 с.

Также обратите внимание, что по мере уточнения запроса с использованием доступных тегов область поиска сужается все больше. В число предлагаемых тегов входят только те, которые соответствуют запросам, зафиксированным отображаемыми метриками. В этом случае теги `exception` и `method` имеют только одно значение; очевидно, что все 30 запросов были запросами GET, и все они привели к ошибке 404 из-за исключения `ResponseStatusException`.

Навигация по конечной точке `/metrics` может быть сложной задачей, но, немного попрактиковавшись, вы без особого труда сможете получить нужную информацию. В главе 16 вы увидите, как Spring Boot Admin упрощает использование данных из конечной точки `/metrics`.

Информация, возвращаемая конечными точками Actuator, помогает получить довольно полное представление о внутренней работе приложения Spring Boot, но она плохо подходит для анализа человеком. Конечные точки Actuator являются конечными точками REST, и предоставляемые ими данные предназначены для использования

другими приложениями, например пользовательским интерфейсом. Учитывая это, давайте посмотрим, как можно представить информацию, предоставляемую Actuator, в удобном веб-приложении.

15.3 Настройка Actuator

Одна из замечательных особенностей Actuator состоит в возможности настройки под конкретные потребности. Некоторые конечные точки поддерживают дополнительные настройки. Однако Actuator позволяет также создавать пользовательские конечные точки.

Рассмотрим несколько способов настройки Actuator, начиная со способов добавления информации в конечную точку `/info`.

15.3.1 Добавление информации в конечную точку `/info`

Как было показано в разделе 15.2.1, конечная точка `/info` изначально не возвращает никакой полезной информации. Но мы легко можем добавить в нее данные, создав свойства с префиксом `info`. Это очень простой способ добавить свои данные в конечную точку `/info`, но не единственный. Spring Boot предлагает интерфейс `InfoContributor`, позволяющий программно добавлять любую информацию в ответ конечной точки `/info`. Spring Boot даже включает пару полезных реализаций `InfoContributor`, которые, несомненно, пригодятся вам.

Давайте посмотрим, как написать свою реализацию `InfoContributor` и добавить некоторую информацию в конечную точку `/info`.

Создание своей реализации `InfoContributor`

Предположим, мы решили добавить в конечную точку `/info` простую статистику по Taco Cloud, например информацию о количестве приготовленных тако. Для этого можно написать класс, реализующий интерфейс `InfoContributor`, внедрить его с помощью `TacoRepository` и затем опубликовать любые статистики, предоставляемые репозиторием `TacoRepository`. В листинге 15.3 показано, как реализовать такой класс.

Листинг 15.3 Реализация интерфейса `InfoContributor`

```
package tacos.actuator;

import java.util.HashMap;
import java.util.Map;

import org.springframework.boot.actuate.info.Info.Builder;
import org.springframework.boot.actuate.info.InfoContributor;
import org.springframework.stereotype.Component;

import tacos.data.TacoRepository;
```

```

@Component
public class TacoCountInfoContributor implements InfoContributor {

    private TacoRepository tacoRepo;

    public TacoCountInfoContributor(TacoRepository tacoRepo) {
        this.tacoRepo = tacoRepo;
    }

    @Override
    public void contribute(Builder builder) {
        long tacoCount = tacoRepo.count().block();
        Map<String, Object> tacoMap = new HashMap<String, Object>();
        tacoMap.put("count", tacoCount);
        builder.withDetail("taco-stats", tacoMap);
    }
}

```

Класс `TacoCountInfoContributor` должен реализовать метод `contribute()`, принимающий объект `Builder` и вызывающий его метод `withDetail()` для добавления информации. В вашей реализации мы обращаемся к `TacoRepository`, вызывая его метод `count()`, чтобы узнать, сколько тачо было приготовлено. В данном конкретном случае мы работаем с реактивным репозиторием, поэтому вызываем `block()`, чтобы получить счетчик из возвращаемого потока `Mono<Long>`. Затем помещаем этот счетчик в ассоциативный массив, который потом передаем построителю с меткой `taco-stats`. После этого конечная точка `/info` будет возвращать количество приготовленных тако:

```

{
    "taco-stats": {
        "count": 44
    }
}

```

Как видите, реализация `InfoContributor` может использовать любые средства, необходимые для предоставления информации. Этим данный подход отличается от создания простых свойств с префиксом `info.`, который ограничен статическими значениями.

ВНЕДРЕНИЕ ИНФОРМАЦИИ О СБОРКЕ В КОНЕЧНУЮ ТОЧКУ /INFO

Spring Boot включает несколько встроенных реализаций `InfoContributor`, которые автоматически добавляют информацию в конечную точку `/info`. Среди них – `BuildInfoContributor`, добавляющий информацию из файла сборки проекта, в том числе версию проекта, отметку времени сборки, а также хост и имя пользователя, выполнившего сборку.

Чтобы добавить информацию о сборке в конечную точку `/info`, добавьте цель `build-info` в раздел `execution` для плагина `Spring Boot Maven Plugin`:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
      <executions>
        <execution>
          <goals>
            <goal>build-info</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Если для сборки проекта используется Gradle, то можно просто добавить следующие строки в файл *build.gradle*:

```
springBoot {
  buildInfo()
}
```

В любом случае система сборки создаст файл с именем *build-info.properties* в файле JAR или WAR, который *BuildInfoContributor* будет читать и добавлять в конечную точку */info*. В следующем фрагменте ответа конечной точки */info* показано, как выглядит эта информация о сборке:

```
{
  "build": {
    "artifact": "tacocloud",
    "name": "taco-cloud",
    "time": "2021-08-08T23:55:16.379Z",
    "version": "0.0.15-SNAPSHOT",
    "group": "sia"
  },
}
```

Эта информация пригодится для точного понимания, какая версия приложения работает и когда она была собрана. Выполнив запрос GET к конечной точке */info*, можно узнать, используется ли у вас последняя и лучшая сборка проекта.

ПРЕДОСТАВЛЕНИЕ ИНФОРМАЦИИ ИЗ РЕПОЗИТОРИЯ GIT

Допустим, что наш проект хранится в Git – системе управления исходным кодом. В этом случае есть возможность включить информацию о коммите Git в конечную точку */info*. Для этого нужно добавить следующий плагин в файл *pom.xml* проекта Maven:

```
<build>
  <plugins>
  ...
    <plugin>
      <groupId>pl.project13.maven</groupId>
      <artifactId>git-commit-id-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Если для сборки проекта используется Gradle, то можно просто добавить следующие строки в файл *build.gradle*:

```
plugins {
  id "com.gorylenko.gradle-git-properties" version "2.3.1"
}
```

Оба плагина делают одно и то же: генерируют артефакт времени сборки с именем *git.properties*, который содержит все метаданные о проекте из Git. Этот файл обслуживает специальная реализация *InfoContributor* и экспортирует его содержимое через конечную точку */info*.

Конечно, для создания файла *git.properties* проект должен иметь метаданные из Git. То есть это должен быть клон репозитория Git или только что инициализированный локальный репозиторий Git с хотя бы одним коммитом. Иначе эти плагины просто не будут работать. Однако вы можете настроить их так, чтобы они игнорировали отсутствующие метаданные Git. Для плагина Maven присвойте свойству *failOnNoGitDirectory* значение *false*:

```
<build>
  <plugins>
  ...
    <plugin>
      <groupId>pl.project13.maven</groupId>
      <artifactId>git-commit-id-plugin</artifactId>
      <configuration>
        <failOnNoGitDirectory>false</failOnNoGitDirectory>
      </configuration>
    </plugin>
  </plugins>
</build>
```

Точно так же можно установить свойство *failOnNoGitDirectory* в системе сборки Gradle, указав его в *gitProperties*:

```
gitProperties {
  failOnNoGitDirectory = false
}
```

В простейшей форме информация из Git, представленная в конечной точке */info*, включает ветвь *git*, хеш коммита и отметку времени, когда было собрано приложение:

```
{  
    "git": {  
        "branch": "main",  
        "commit": {  
            "id": "df45505",  
            "time": "2021-08-08T21:51:12Z"  
        }  
    },  
    ...  
}
```

Эта информация достаточно точно описывает состояние кода на момент сборки проекта. Но, присвоив свойству `management.info.git.mode` значение `full`, можно получить очень подробную информацию о коммите Git, послужившем основой для сборки проекта:

```
management:  
  info:  
    git:  
      mode: full
```

В листинге 15.4 показан пример сведений, возвращаемых после присваивания свойству `management.info.git.mode` значения `full`.

Листинг 15.4 Полная информация о коммите Git, возвращаемая конечной точкой /info

```
"git": {  
    "local": {  
        "branch": {  
            "ahead": "8",  
            "behind": "0"  
        }  
    },  
    "commit": {  
        "id": {  
            "describe-short": "df45505-dirty",  
            "abbrev": "df45505",  
            "full": "df45505daaf3b1347b0ad1d9dca4ebbc6067810",  
            "describe": "df45505-dirty"  
        },  
        "message": {  
            "short": "Apply chapter 18 edits",  
            "full": "Apply chapter 18 edits"  
        },  
        "user": {  
            "name": "Craig Walls",  
            "email": "craig@habuma.com"  
        },  
        "author": {  
            "time": "2021-08-08T15:51:12-0600"  
        },  
        "committer": {  
            "time": "2021-08-08T15:51:12-0600"  
        }  
    }  
}
```

```

        "time": "2021-08-08T15:51:12-0600"
    },
    "time": "2021-08-08T21:51:12Z"
},
"branch": "master",
"build": {
    "time": "2021-08-09T00:13:37Z",
    "version": "0.0.15-SNAPSHOT",
    "host": "Craigs-MacBook-Pro.local",
    "user": {
        "name": "Craig Walls",
        "email": "craig@habuma.com"
    }
},
"tags": "",
"total": {
    "commit": {
        "count": "196"
    }
},
"closest": {
    "tag": {
        "commit": {
            "count": ""
        },
        "name": ""
    }
},
"remote": {
    "origin": {
        "url": "git@github.com:habuma/spring-in-action-6-samples.git"
    }
},
"dirty": "true"
},

```

В дополнение к отметке времени и сокращенному хешу коммита полная версия включает имя и адрес электронной почты пользователя, выполнившего коммит, а также сообщение в коммите и другую информацию, позволяющую точно определить, какой код использовался для сборки проекта. Обратите также внимание, что поле `dirty` в листинге 15.4 имеет значение `true`. Так отмечается наличие некоторых незафиксированных изменений в каталоге сборки на момент компиляции проекта. Трудно представить более полную информацию, чем эта!

15.3.2 Определение своих индикаторов работоспособности

В Spring Boot имеется несколько готовых индикаторов, предоставляющих информацию о работоспособности для многих распространенных внешних систем, с которыми может интегрироваться приложение

Spring. Но в какой-то момент можно обнаружить, что взаимодействия с какой-то внешней системой Spring Boot неофициально не поддерживает и не имеет индикатора, оценивающего ее работоспособность.

Например, приложение может интегрироваться с некоторой устаревшей системой, и его работоспособность может зависеть от работоспособности этой системы. Чтобы создать свой индикатор работоспособности, нужно определить bean-компонент, реализующий интерфейс `HealthIndicator`.

Как оказывается, облачным службам Taco не нужен нестандартный индикатор работоспособности, потому что готовых, имеющихся в Spring Boot, более чем достаточно. Но чтобы показать создание собственного индикатора работоспособности, рассмотрим листинг 15.5, где представлена простая реализация `HealthIndicator`, определяющая работоспособность некоторым случайным образом в зависимости от времени суток.

Листинг 15.5 Необычная реализация `HealthIndicator`

```
package tacos.actuator;

import java.util.Calendar;
import org.springframework.boot.actuate.health.Health;
import org.springframework.boot.actuate.health.HealthIndicator;
import org.springframework.stereotype.Component;

@Component
public class WackoHealthIndicator
    implements HealthIndicator {

    @Override
    public Health health() {
        int hour = Calendar.getInstance().get(Calendar.HOUR_OF_DAY);
        if (hour > 12) {
            return Health
                .outOfService()
                .withDetail("reason", "I'm out of service after lunchtime")
                .withDetail("hour", hour)
                .build();
        }
        if (Math.random() <= 0.1) {
            return Health
                .down()
                .withDetail("reason", "I break 10% of the time")
                .build();
        }
        return Health
            .up()
            .withDetail("reason", "All is good!")
            .build();
    }
}
```

Этот необычный индикатор работоспособности сначала проверяет текущее время, и если полдень уже прошел, то возвращает статус работоспособности OUT_OF_SERVICE с некоторыми подробностями, объясняющими причину этого статуса. В период до полудня есть 10%-ный шанс, что индикатор сообщит статус DOWN, потому что использует случайное число, выбирая между статусами UP и DOWN. Если случайное число меньше 0,1, то индикатор вернет статус DOWN, иначе – статус UP.

Очевидно, что индикатор работоспособности в листинге 15.5 бесполезен в реальных приложениях. Но представьте, что вместо проверки текущего времени или случайного числа индикатор посыпает запрос некоторой удаленной системе и определяет ее статус на основе полученного ответа. В таком случае это был бы очень полезный индикатор работоспособности.

15.3.3 Регистрация пользовательских метрик

В разделе 15.2.4 мы видели, как пользоваться конечной точкой /metrics и получать информацию о различных метриках, опубликованных в Actuator, особое внимание уделив метрикам, связанным с HTTP-запросами. Метрики, поддерживаемые в Actuator, очень полезны, но конечная точка /metrics не ограничивается только встроенными метриками.

В конечном счете метрики в Actuator реализуются независимым от поставщика фасадом метрик Micrometer (<https://micrometer.io/>), который позволяет приложениям публиковать любые нужные им метрики для отображения с помощью сторонних систем мониторинга, включая Prometheus, Datadog и New Relic.

Самый простой способ публикации метрик с помощью Micrometer – использовать компонент MeterRegistry из Micrometer. В приложении Spring Boot для этого достаточно внедрить MeterRegistry везде, где может понадобиться опубликовать счетчики, таймеры или датчики, фиксирующие значения метрик вашего приложения.

В качестве примера публикации своих метрик предположим, что мы решили хранить счетчики тако, приготовленных из разных ингредиентов, т. е. сколько было приготовлено тако с листьями салата, говяжьей котлетой, из мучных лепешек или с любыми другими доступными ингредиентами. Компонент TacoMetrics в листинге 15.6 показывает, как можно использовать MeterRegistry для сбора этой информации.

Листинг 15.6 Компонент TacoMetrics, регистрирующий метрики со счетчиками ингредиентов

```
package tacos.actuator;

import java.util.List;
import org.springframework.data.rest.core.event.AbstractRepositoryEventListener;
import org.springframework.stereotype.Component;
```

```
import io.micrometer.core.instrument.MeterRegistry;
import tacos.Ingredient;
import tacos.Taco;

@Component
public class TacoMetrics extends AbstractRepositoryEventListener<Taco> {

    private MeterRegistry meterRegistry;

    public TacoMetrics(MeterRegistry meterRegistry) {
        this.meterRegistry = meterRegistry;
    }

    @Override
    protected void onAfterCreate(Taco taco) {
        List<Ingredient> ingredients = taco.getIngredients();
        for (Ingredient ingredient : ingredients) {
            meterRegistry.counter("tacocloud",
                "ingredient", ingredient.getId()).increment();
        }
    }
}
```

Как видите, `TacoMetrics` внедряется посредством своего конструктора с помощью `MeterRegistry`. Он также наследует `AbstractRepositoryEventListener` – класс Spring Data, позволяющий перехватывать события репозиториев, и переопределяет метод `onAfterCreate()`, чтобы получать уведомления всякий раз, когда сохраняется новый объект `Taco`.

Метод `onAfterCreate()` обновляет счетчик каждого ингредиента, тег которого совпадает с идентификатором ингредиента. Если счетчик с таким тегом уже существует, он будет использован повторно. Вызов метода `counter` увеличивает счетчик, указывая, что был создан еще один `taco` с этим ингредиентом.

Создав несколько `taco`, можно попробовать запросить конечную точку `/metrics`, чтобы получить счетчики ингредиентов. Запрос GET к `/metrics/tacocloud` вернет некоторые метрики, как показано ниже:

```
$ curl localhost:8080/actuator/metrics/tacocloud
{
  "name": "tacocloud",
  "measurements": [ { "statistic": "COUNT", "value": 84 } ],
  "availableTags": [
    {
      "tag": "ingredient",
      "values": [ "FLTO", "CHED", "LETC", "GRBF",
                  "COTO", "JACK", "TMT", "SLSA" ]
    }
  ]
}
```

Значение счетчика COUNT в разделе `measurements` здесь мало что значит, потому что это сумма всех счетчиков всех ингредиентов. Но давайте предположим, что нам нужно узнать, сколько тако было приготовлено из мучных лепешек (FLTO). Для этого нужно передать в запросе тег `ingredient` со значением FLTO:

```
$ curl localhost:8080/actuator/metrics/tacocloud?tag=ingredient:FLTO
{
  "name": "tacocloud",
  "measurements": [
    { "statistic": "COUNT", "value": 39 }
  ],
  "availableTags": []
}
```

Теперь ясно, что 39 тако были приготовлены из мучных лепешек.

15.3.4 Создание пользовательских конечных точек

На первый взгляд может показаться, что конечные точки Actuator реализованы как обычные контроллеры Spring MVC. Но, как вы увидите в главе 17, они могут быть реализованы также как компоненты JMX MBean, а еще через HTTP-запросы. Следовательно, эти конечные точки должны быть чем-то большим, чем простой класс контроллера.

На самом деле конечные точки Actuator определяются совершенно иначе, чем контроллеры. Вместо класса с аннотацией `@Controller` или `@RestController` конечные точки Actuator определяются с помощью классов с аннотацией `@Endpoint`.

Более того, вместо использования аннотаций поддержки HTTP, таких как `@GetMapping`, `@PostMapping` или `@DeleteMapping`, операции конечной точки Actuator определяются методами с аннотациями `@ReadOperation`, `@WriteOperation` и `@DeleteOperation`. Эти аннотации не подразумевают какого-то конкретного механизма взаимодействий и фактически позволяют Actuator обмениваться данными с использованием любых механизмов, таких как HTTP и JMX. Чтобы показать, как написать свою конечную точку Actuator, рассмотрим класс `NotesEndpoint` в листинге 15.7.

Листинг 15.7 Нестандартная конечная точка для управления заметками

```
package tacos.actuator;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import org.springframework.boot.actuate.endpoint.annotation.DeleteOperation;
import org.springframework.boot.actuate.endpoint.annotation.Endpoint;
import org.springframework.boot.actuate.endpoint.annotation.ReadOperation;
import org.springframework.boot.actuate.endpoint.annotation.WriteOperation;
```

```
import org.springframework.stereotype.Component;

@Component
@Endpoint(id="notes", enableByDefault=true)
public class NotesEndpoint {

    private List<Note> notes = new ArrayList<>();

    @ReadOperation
    public List<Note> notes() {
        return notes;
    }

    @WriteOperation
    public List<Note> addNote(String text) {
        notes.add(new Note(text));
        return notes;
    }

    @DeleteOperation
    public List<Note> deleteNote(int index) {
        if (index < notes.size()) {
            notes.remove(index);
        }
        return notes;
    }

    class Note {
        private Date time = new Date();
        private final String text;

        public Note(String text) {
            this.text = text;
        }

        public Date getTime() {
            return time;
        }

        public String getText() {
            return text;
        }
    }
}
```

Это простая конечная точка для создания заметок, с помощью которой можно создать новую и удалить имеющуюся заметку или прочитать список заметок. Готов признать, что эта конечная точка не особенно полезна в качестве конечной точки Actuator. Но, учитывая богатое разнообразие готовых конечных точек Actuator, мне было трудно придумать более практический пример пользовательской конечной точки для Actuator.

Как бы то ни было, класс `NotesEndpoint` снабжен аннотацией `@Component`, поэтому он будет обнаружен на этапе сканирования компонентов и на его основе Spring создаст bean-компонент в контексте приложения. Но более важно для нашего обсуждения, что он также отмечен аннотацией `@Endpoint`, превращающей его в конечную точку Actuator с идентификатором `notes`. Кроме того, он включен по умолчанию, поэтому его не требуется включать явно в конфигурационном свойстве `management.web.endpoints.web.exposure.include`.

Как видите, `NotesEndpoint` предлагает по одной операции каждого типа:

- метод `notes()` с аннотацией `@ReadOperation` возвращает список доступных заметок. С точки зрения HTTP это означает, что он будет обрабатывать HTTP-запросы GET к конечной точке `/actuator/notes` и возвращать список заметок в формате JSON;
- метод `addNote()` с аннотацией `@WriteOperation` создает новую заметку на основе заданного текста и добавляет ее в список. С точки зрения HTTP это означает, что он будет обрабатывать HTTP-запросы POST, содержащие в теле объект JSON со свойством `text`. В ответ он возвращает текущее состояние списка заметок;
- метод `deleteNote()` с аннотацией `@DeleteOperation` удаляет заметку с указанным индексом. С точки зрения HTTP это означает, что он будет обрабатывать HTTP-запросы DELETE, содержащие индекс в параметре запроса.

Чтобы поэкспериментировать с этой конечной точкой и увидеть ее в действии, можно использовать утилиту `curl`. Для начала добавим пару заметок, послав два отдельных запроса POST:

```
$ curl localhost:8080/actuator/notes \
  -d'{"text":"Bring home milk"}' \
  -H"Content-type: application/json"
[{"time":"2020-06-08T13:50:45.085+0000","text":"Bring home milk"}]

$ curl localhost:8080/actuator/notes \
  -d'{"text":"Take dry cleaning"}' \
  -H"Content-type: application/json"
[{"time":"2021-07-03T12:39:13.058+0000","text":"Bring home milk"}, \
 {"time":"2021-07-03T12:39:16.012+0000","text":"Take dry cleaning"}]
```

Как видите, каждый раз, когда публикуется новая заметка, конечная точка отвечает обновленным списком заметок. Но если позже понадобится просмотреть список заметок, то можно выполнить простой запрос GET:

```
$ curl localhost:8080/actuator/notes
[{"time":"2021-07-03T12:39:13.058+0000","text":"Bring home milk"}, \
 {"time":"2021-07-03T12:39:16.012+0000","text":"Take dry cleaning"}]
```

Чтобы удалить одну из заметок, нужно выполнить запрос DELETE и передать в параметре индекс удаляемой заметки:

```
$ curl localhost:8080/actuator/notes?index=1 -X DELETE
[{"time":"2021-07-03T12:39:13.058+0000","text":"Bring home milk"}]
```

Важно отметить, что даже притом, что я показал, только как взаимодействовать с конечной точкой с помощью HTTP, она также будет доступна как компонент JMX MBean, доступ к которому можно получить с помощью любого JMX-клиента. Но если вы решите ограничить доступность конечных точек лишь протоколом HTTP, то замените аннотацию `@Endpoint` на `@WebEndpoint`:

```
@Component
@WebEndpoint(id="notes", enableByDefault=true)
public class NotesEndpoint {
    ...
}
```

Аналогично, если вы предпочтете разрешить доступ к конечной точке только для MBean, то используйте аннотацию `@JmxEndpoint`.

15.4 Защита конечных точек Actuator

Информация, представляемая через конечные точки Actuator, может содержать конфиденциальные сведения, не предназначенные для посторонних глаз. Кроме того, поскольку Actuator предоставляет операции, позволяющие изменять свойства окружения и уровни журналирования, было бы неплохо защитить доступ к конечным точкам Actuator, чтобы только клиенты с надлежащими привилегиями могли использовать их.

Безопасность конечных точек Actuator, конечно, важна, но ее обеспечение не входит в обязанности Actuator. Поэтому мы должны использовать Spring Security для защиты Actuator. А так как конечные точки Actuator – это просто пути в приложении, как любые другие, в организации защиты Actuator нет ничего уникального. Все, о чем рассказывалось в главе 5, в равной степени применимо к конечным точкам Actuator.

Пути ко всем конечным точкам Actuator начинаются с общего префикса `/actuator` (или другого, если вы определите свой базовый путь в свойстве `management.endpoints.web.base-path`), поэтому ко всем конечным точкам Actuator легко применить правила авторизации. Например, чтобы потребовать наличие у пользователя привилегии `ROLE_ADMIN`, можно переопределить метод `configure()` класса `WebSecurityConfigurerAdapter`:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
        .antMatchers("/actuator/**").hasRole("ADMIN")
```

```

    .and()
    .httpBasic();
}

```

После этого обрабатываться будут только запросы, поступившие от аутентифицированного пользователя, обладающего привилегией `ROLE_ADMIN`. Здесь также настраивается базовая аутентификация HTTP, чтобы клиентские приложения могли отправлять закодированные данные аутентификации в заголовках `Authorization` запросов.

Единственная настоящая проблема с защитой Actuator при таком подходе заключается в том, что путь к конечным точкам жестко за-программирован как `/actuator/**`. Если он изменится из-за настройки свойства `management.endpoints.web.base-path`, то защита перестанет работать. Чтобы решить проблему, Spring Boot предоставляет `EndpointRequest` – класс сопоставления запросов, который устраняет зависимость от конкретного пути. Используя `EndpointRequest`, можно применить те же требования безопасности к конечным точкам Actuator, независимо от заданного базового пути к конечным точкам Actuator:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .requestMatcher(EndpointRequest.toAnyEndpoint())
        .authorizeRequests()
        .anyRequest().hasRole("ADMIN")
        .and()
        .httpBasic();
}

```

Метод `EndpointRequest.toAnyEndpoint()` возвращает объект, реализующий проверку запросов на принадлежность к конечным точкам Actuator. Если понадобится исключить какие-то конечные точки из проверки, то можно вызвать метод `excluding()`, передав ему имена точек:

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    http
        .requestMatcher(EndpointRequest.toAnyEndpoint())
        .excluding("health", "info")
        .authorizeRequests()
        .anyRequest().hasRole("ADMIN")
        .and()
        .httpBasic();
}

```

Напротив, если потребуется защитить только выбранные конечные точки Actuator, то вместо `toAnyEndpoint()` можно вызвать метод `to()` и передать ему имена выбранных конечных точек, например:

```
@Override  
protected void configure(HttpSecurity http) throws Exception {  
    http  
        .requestMatcher(EndpointRequest.to("beans", "threaddump", "loggers"))  
        .authorizeRequests()  
            .anyRequest().hasRole("ADMIN")  
            .and()  
            .httpBasic();  
}
```

В этом случае ограничения доступа затронут лишь конечные точки `/beans`, `/threaddump` и `/loggers`. Все остальные конечные точки Actuator остаются доступными всем.

Итоги

- Spring Boot Actuator реализует множество конечных точек, доступных через HTTP или как компоненты JMX MBean. Они позволяют заглянуть внутрь приложения Spring Boot.
- Большинство конечных точек Actuator отключены по умолчанию, но их можно выборочно включать и выключать с помощью свойств `management.endpoints.web.exposure.include` и `management.endpoints.web.exposure.exclude`.
- Некоторые конечные точки, такие как `/loggers` и `/env`, поддерживают операции записи, что дает возможность изменять конфигурацию работающего приложения на лету.
- Через конечную точку `/info` можно опубликовать подробную информацию о сборке приложения и коммите Git.
- Проверять работоспособность приложения можно с помощью собственных индикаторов, отслеживающих работоспособность самого приложения и внешних зависимостей.
- С помощью Micrometer можно зарегистрировать пользовательские метрики, что обеспечит бесшовную интеграцию приложений Spring Boot со многими популярными механизмами сбора метрик, такими как Datadog, New Relic и Prometheus.
- Конечные точки Actuator, доступные через HTTP, можно защитить с помощью фреймворка Spring Security, как и любые другие конечные точки в приложении Spring.

16 Администрирование Spring

В этой главе рассматриваются следующие темы:

- настройка Spring Boot Admin;
- регистрация клиентских приложений;
- работа с конечными точками Actuator;
- защита Admin Server.

Картинка стоит тысячи слов (так говорят), и для многих пользователей одно удобное веб-приложение стоит тысячи вызовов API. Не поймите меня превратно, я фанат командной строки и большой поклонник `curl` и `HTTPie` для работы с REST API. Но иногда ввод команды вручную для вызова конечной точки REST с последующим анализом результатов может быть менее эффективным, чем простой щелчок на ссылке и чтение результатов в веб-браузере.

В предыдущей главе мы познакомились с конечными точками Spring Boot Actuator. Поскольку эти конечные точки возвращают ответы в формате JSON, их можно использовать без ограничений. В этой главе мы посмотрим, как организовать внешний пользовательский интерфейс для отображения информации из Actuator, чтобы упростить ее анализ, а также для сбора оперативных данных, которые трудно получить напрямую из Actuator.

16.1 Spring Boot Admin

Меня иногда спрашивают, имеет ли это смысл и насколько сложно разработать веб-приложение, использующее конечные точки Actuator и отображающее их содержимое в удобном пользовательском интерфейсе. В ответ я говорю, что это всего лишь REST API, а значит, все возможно. Но зачем создавать свой пользовательский интерфейс для Actuator, если хорошие ребята из компании Codecentric AG (<https://www.codecentric.de/>), базирующейся в Германии, уже сделали всю работу за нас?

Spring Boot Admin – это веб-приложение для администрирования, упрощающее работу с конечными точками Actuator. Приложение разделено на два основных компонента: сервер Spring Boot Admin и клиенты. Сервер собирает данные из Actuator, которые передаются ему из одного или нескольких приложений Spring Boot, являющихся клиентами Spring Boot Admin, как показано на рис. 16.1.

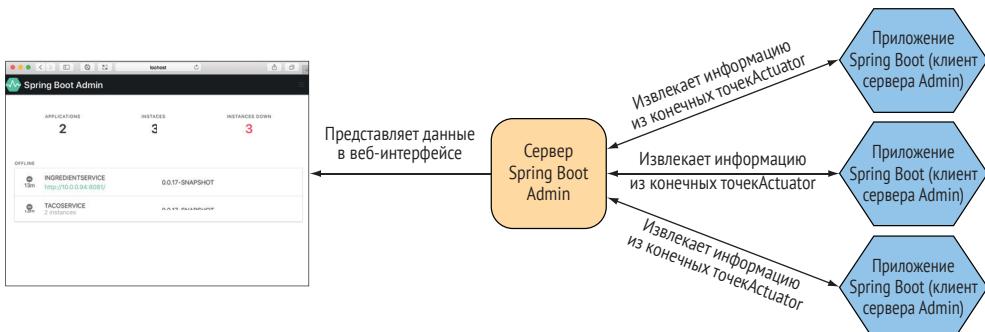


Рис. 16.1 Сервер Spring Boot Admin использует конечные точки Actuator в одном или нескольких приложениях Spring Boot и отображает данные в пользовательском веб-интерфейсе

Каждое приложение-клиент нужно зарегистрировать на сервере Spring Boot Admin, но прежде следует настроить сервер Spring Boot Admin для получения информации из конечных точек Actuator каждого клиента.

16.1.1 Создание сервера Admin

Чтобы запустить сервер Admin, необходимо создать новое приложение Spring Boot и добавить зависимость от сервера Admin в спецификацию сборки проекта. Сервер Admin обычно действует как автономное приложение, отдельное от любых других приложений. Поэтому самый простой способ создать его – использовать приложение Spring Boot Initializr для создания нового проекта Spring Boot и установить флагок **Spring Boot Admin (Server)**. В результате в блок <dependencies> будет включена следующая зависимость:

```
<dependency>
<groupId>de.codecentric</groupId>
```

```
<artifactId>spring-boot-admin-starter-server</artifactId>
</dependency>
```

Затем вам нужно снабдить основной класс конфигурации аннотацией `@EnableAdminServer`, как показано ниже:

```
package tacos.admin;

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

import de.codecentric.boot.admin.server.config.EnableAdminServer;

@EnableAdminServer
@SpringBootApplication
public class AdminServerApplication {

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

Наконец, поскольку сервер Admin будет не единственным приложением, работающим локально, его следует настроить на прослушивание уникального порта, к которому легко получить доступ (например, любой свободный порт с номером больше 1024). Для этого примера я выбрал порт 9090:

```
server:
  port: 9090
```

Теперь сервер Admin готов к работе. Если запустить его прямо сейчас и открыть в браузере страницу `http://localhost:9090`, то вы увидите страницу, показанную на рис. 16.2.

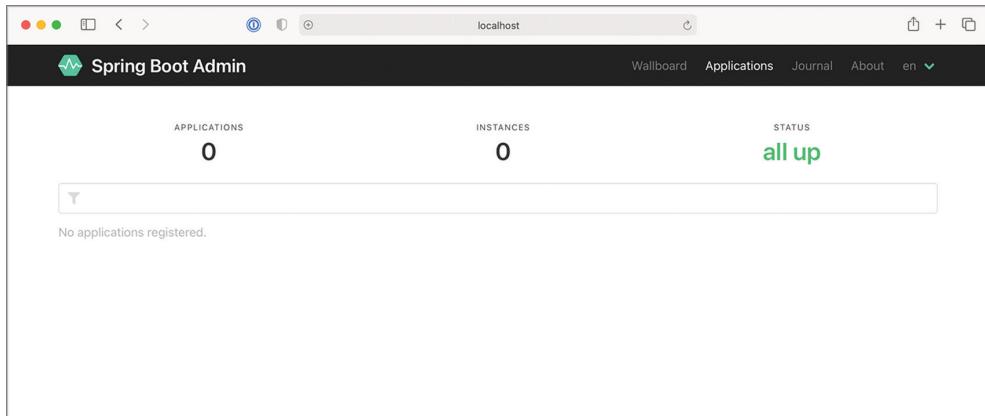


Рис. 16.2 Только что созданный сервер отображает чистый веб-интерфейс Spring Boot Admin. Никаких приложений пока не зарегистрировано

Как видите, Spring Boot Admin показывает, что запущено ноль экземпляров клиентских приложений. Это бессмысленная информация, если учесть, что ниже выводится сообщение, в котором говорится, что пока нет зарегистрированных приложений. Чтобы получить пользу от сервера Admin, необходимо зарегистрировать в нем какие-нибудь приложения.

16.1.2 Регистрация клиентов сервера Admin

Поскольку сервер Admin – это приложение, работающее независимо от других приложений Spring Boot, откуда он должен извлекать информацию для отображения, нужно каким-то образом информировать сервер Admin об этих приложениях. Вот два способа регистрации клиентов Spring Boot Admin:

- явная регистрация каждого приложения на сервере Admin;
- получение информации о клиентских приложениях из службы реестра Eureka.

Далее мы рассмотрим первый способ регистрации приложений на сервере Spring Boot Admin. А более подробную информацию о втором способе регистрации с помощью Eureka вы найдете в документации Spring Cloud, доступной по адресу <https://docs.spring.io/spring-cloud-netflix/docs/current/reference/html/>, или в книге «Spring Microservices in Action, 2nd Edition», написанной Джоном Карнеллом (John Carnell) и Иллари Уайлупо Санчесом (Illyari Huaylupo Sánchez)¹.

Чтобы приложение Spring Boot зарегистрировало себя в качестве клиента на сервере Admin, в его спецификацию сборки нужно включить начальную зависимость от клиента Spring Boot Admin. Эту зависимость легко добавить, установив флагок **Spring Boot Admin (Client)** в приложении Initializr или добавив следующий элемент `<dependency>` в файл сборки Maven:

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
</dependency>
```

После добавления в проект клиентской библиотеки также необходимо задать местоположение сервера Admin, чтобы клиент мог зарегистрироваться. Для этого следует определить свойство `spring.boot.admin.client.url` с корневым URL сервера Admin:

```
spring:
  boot:
    admin:
      client:
        url: http://localhost:9090
```

¹ Карнелл Дж., Санчес И. У. Микросервисы Spring в действии. ДМК-Пресс, 2021, ISBN: 978-5-97060-971-2. – Прим. перев.

Обратите внимание, что также должно быть установлено свойство `spring.application.name`. Это свойство используется несколькими проектами Spring для идентификации приложений. Значение этого свойства будет передаваться серверу Admin и отображаться в веб-интерфейсе в качестве метки вместе с информацией о приложении.

Информации о приложении Taco Cloud, как показано на рис. 16.3, не так много, и все же мы можем видеть продолжительность безотказной работы приложения, настроена ли цель `build-info` для плагина Spring Boot Maven (как обсуждалось в разделе 15.3.1) и версию сборки. Но имейте в виду, что после щелчка на ссылке с именем приложения сервер Admin покажет вам массу других сведений о приложении.

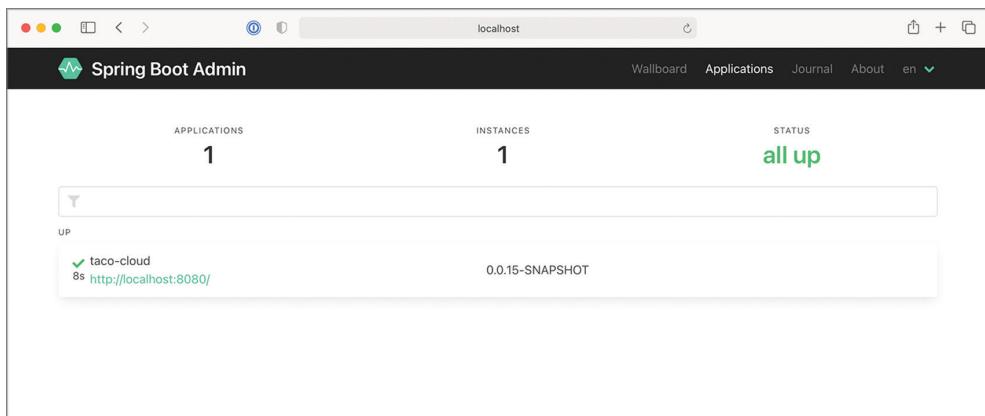


Рис. 16.3 Веб-интерфейс Spring Boot Admin отображает единственное зарегистрированное приложение

Теперь, когда мы зарегистрировали приложение Taco Cloud на сервере Admin, давайте посмотрим, что может предложить сервер.

16.2 Исследование возможностей сервера Admin

После регистрации всех приложений Spring Boot в качестве клиентов сервер Admin начинает собирать массив информации, характеризующий происходящее внутри каждого приложения, включая:

- общее состояние работоспособности;
- любые метрики, опубликованные через Micrometer и конечную точку `/metrics`;
- свойства окружения;
- уровни журналирования для пакетов и классов.

На самом деле сервер собирает почти всю информацию, которая доступна через конечные точки Actuator, и отображает ее в удобном

для человека формате, формируя графики и предоставляя фильтры. Объем информации, доступной в веб-интерфейсе сервера Admin, намного больше, чем мы сможем рассмотреть в этой главе. Поэтому в оставшейся части данного раздела я охвачу лишь основные аспекты сервера Admin.

16.2.1 Обзор общего состояния приложения

Как отмечалось в главе 15, самая основная часть информации, предоставляемой Actuator, – это информация о работоспособности приложения и о нем самом, доступная через конечные точки `/health` и `/info`. Сервер Admin отображает эту информацию в разделе **Details** (Подробности), как показано на рис. 16.4.

The screenshot shows the Spring Boot Admin interface for the application `taco-cloud` with ID `a973ff14be49`. The left sidebar has a green header bar with the application name and ID, followed by a list of monitoring categories: Insights, Details (which is selected and highlighted in green), Metrics, Environment, Beans, Configuration Properties, Scheduled Tasks, Loggers, Mappings, and Caches. The main content area is divided into two sections: **Info** and **Health**.

Info section:

- contact**: email: support@tacocloud.com, phone: 822-625-6831
- git**: local: branch: ahead: 8, behind: 0, commit: id: df455055daaf3b1347b0ad1d9dca4ebbc6c, describe-short: df455055-dirty, abbrev: df45505, full: df455055daaf3b1347b0ad1d9dca4ebbc6c, describe: df45505-dirty, message: short: Apply chapter 18 edits

Health section:

	Instance	UP
diskSpace	494 GB	UP
total	263 GB	UP
free	10.5 MB	UP
exists	true	UP
mongo	3.5.5	UP
version		

Below the Health table, there are three links: <http://localhost:8080/>, <http://localhost:8080/actuator>, and <http://localhost:8080/actuator/health>.

Рис. 16.4. В разделе **Details** (Подробности) в веб-интерфейсе Spring Boot Admin отображается общая информация о работоспособности приложения и о нем самом

Если прокрутить вниз таблицы **Health** (Работоспособность) и **Info** (Информация) в разделе **Details** (Подробности), можно найти полезную статистику из JVM приложения, включая графики, отображающие потребление памяти, потоков выполнения и процессора (рис. 16.5).

Информация, отображаемая на графиках, а также сведения в подразделах **Processes** (Процессы) и **Garbage Collection Pauses** (Паузы в сборке мусора) могут помочь понять, как приложение использует ресурсы JVM.

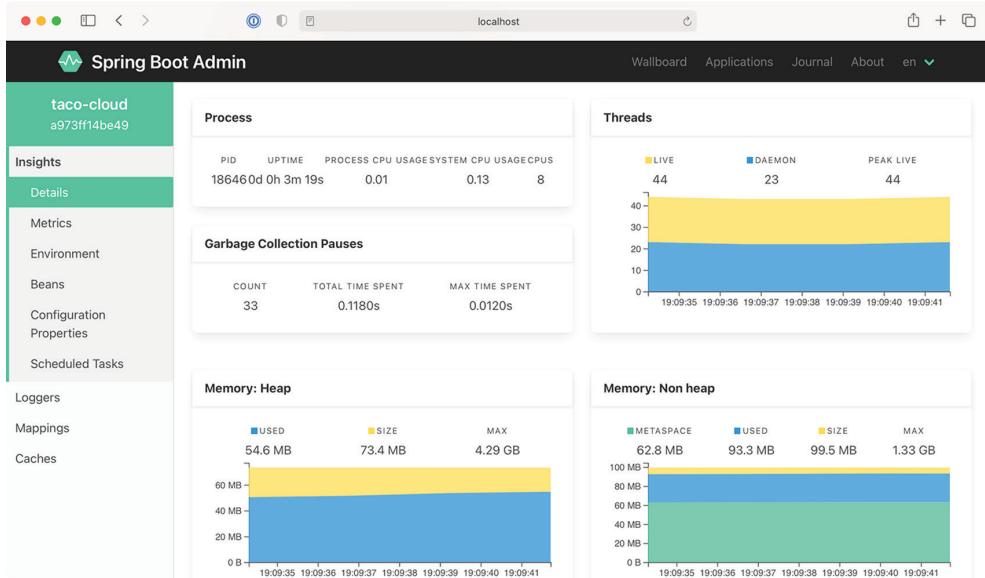


Рис. 16.5 Если прокрутить вниз раздел Details (Подробности), можно увидеть дополнительную информацию из недр JVM, включая потребление памяти, потоков выполнения и процессора

16.2.2 Просмотр ключевых метрик

Информация, возвращаемая конечной точкой `/metrics`, является, по-жалуй, наименее удобочитаемой из всех сведений, предоставляемых конечными точками Actuator. Однако сервер Admin позволяет нам, простым смертным, просматривать и анализировать метрики приложения в разделе **Metrics** (Метрики) веб-интерфейса.

Изначально в разделе **Metrics** (Метрики) не отображается никаких метрик. Но в верхней части страницы имеется форма, заполнив которую, можно настроить отображение одной или нескольких метрик.

На рис. 16.6 я настроил наблюдение за двумя метриками из категории `http.server.requests`. Первая сообщает количество полученных HTTP-запросов GET, на которые был отправлен ответ со статусом 200 (OK). Вторая – количество запросов, на которые был отправлен ответ со статусом HTTP 404 (NOT FOUND).

Что особенно примечательно в этих метриках (да и во всем, что отображается на сервере Admin), – они отображаются в режиме реального времени и обновляются автоматически.

16.2.3 Исследование свойств окружения

Конечная точка Actuator `/env` возвращает все свойства окружения из всех его источников свойств, доступных приложению Spring Boot. Возвращаемый конечной точкой ответ в формате JSON не так уж

сложен для чтения, однако сервер Admin отображает его в разделе **Environment** (Окружение) в гораздо более удобном для восприятия виде, как показано на рис. 16.7.

The screenshot shows the Spring Boot Admin interface on a Mac OS X system. The title bar says "Spring Boot Admin". The left sidebar has a green header "taco-cloud a973ff14be49" and a "Metrics" tab selected. The main area has a search bar with "http.server.requests" and several dropdown filters: "exception" (empty), "method" (GET), "uri" (empty), "outcome" (empty), and "status" (404). A green button "Add Metric" is at the bottom right. Below the filters is a table with columns: http.server.requests, COUNT, TOTAL_TIME, and MAX. Two rows are shown: "method:GET status:200" with COUNT 160, TOTAL_TIME 10.481907661, and MAX 0.373752458; and "method:GET status:404" with COUNT 8, TOTAL_TIME 0.456307709, and MAX 0.186124125. Each row has a trash icon at the end.

Рис. 16.6 В разделе Metrics (Метрики) можно настроить наблюдение за любыми метриками, доступными через конечную точку /metrics

The screenshot shows the Spring Boot Admin interface on a Mac OS X system. The title bar says "Spring Boot Admin". The left sidebar has a green header "taco-cloud a973ff14be49" and an "Environment" tab selected. The main area has a search bar with "spring." and a filter icon. A table lists environment properties: "systemProperties" (empty); "Config resource 'class path resource [application.yml]' via location 'optional:classpath:/' (document #0)" (empty); and three entries from application.yml: "spring.application.name" (value "taco-cloud"), "spring.data.rest.base-path" (value "/api"), and "spring.data.cassandra.keyspace-name" (value "tacocloud"). Each entry includes a timestamp and a "recreate" link.

Рис. 16.7 В разделе Environment (Окружение) отображаются свойства окружения и предоставляется возможность фильтровать и переопределять эти свойства

Количество свойств может исчисляться сотнями, поэтому сервер Admin предлагает возможность отфильтровать список по именам или значениям свойств. На рис. 16.7 показан отфильтрованный список свойства, имена и/или значения которых содержат текст "spring.". Сервер Admin также позволяет создавать или переопределять свойства окружения, предлагая для этого форму под заголовком **Environment Manager** (Диспетчер окружения).

16.2.4 Просмотр и изменение уровней журналирования

Конечная точка `/loggers` помогает узнать и переопределить уровни журналирования в работающем приложении. Раздел **Loggers** (Журналирование) в веб-интерфейсе сервера Admin упрощает управление журналированием в приложении. На рис. 16.8 показан список средств журналирования, отфильтрованных по имени `org.springframework.boot`.

The screenshot shows the Spring Boot Admin interface with the URL `localhost`. On the left, there's a sidebar with tabs: Insights, Loggers (which is selected and highlighted in green), Mappings, and Caches. The main content area has a title bar with a search input containing `org.springframework.boot`, a status indicator `171/715`, and a refresh button. Below this, there's a checkbox for `class only` and another for `configured`. A table lists several package names with their corresponding log levels: `org.springframework.boot` (INFO), `org.springframework.boot.ApplicationServletEnvironment` (INFO), `org.springframework.boot.BeanDefinitionLoader` (INFO), `org.springframework.boot.BeanDefinitionLoader$ClassExcludeFilter` (INFO), `org.springframework.boot.DefaultApplicationArguments` (INFO), `org.springframework.boot.DefaultApplicationArguments$Source` (INFO), and `org.springframework.boot.SpringApplication` (INFO). Each row has buttons for `OFF`, `ERROR`, `WARN`, `INFO` (which is blue and bolded), `DEBUG`, `TRACE`, and `Reset`. At the bottom of the table, there's a footer message: `org.springframework.boot.SpringApplicationShutdownHook`.

Рис. 16.8 В разделе **Loggers** (Журналирование) отображаются уровни журналирования для пакетов и классов в приложении и дается возможность переопределять их

По умолчанию сервер Admin отображает уровни журналирования для всех пакетов и классов. Их можно отфильтровать по именам (только для классов) и/или уровням журналирования, которые настроены явно, а не унаследованы от корневого средства журналирования.

16.3 Защита сервера Admin

Как обсуждалось в предыдущей главе, информация, предоставляемая конечными точками Actuator, не предназначена для общего пользования. Она содержит сведения о приложении, которые должен видеть только администратор приложения. Более того, некоторые конечные точки позволяют вносить изменения, что, безусловно, не должно быть доступно посторонним.

Насколько безопасность важна для Actuator, настолько же она важна для сервера Admin. Более того, если конечные точки Actuator требуют аутентификации, то серверу Admin необходимо знать учетные данные, чтобы получить доступ к этим конечным точкам. Давайте посмотрим, как защитить сервер Admin. Начнем с запроса аутентификации.

16.3.1 Включение регистрации на сервере Admin

Обеспечить дополнительную защиту сервера Admin – хорошая идея, потому что по умолчанию он никак не защищен. Поскольку сервер Admin является приложением Spring Boot, его можно защитить с помощью Spring Security, как и любое другое приложение Spring Boot. Так же как в случае с любым другим приложением, есть возможность выбрать схему защиты, которая лучше всего соответствует вашим потребностям.

Как минимум можно добавить начальную зависимость Spring Boot Security в спецификацию сборки сервера Admin, установив флагок **Security** (Безопасность) в Initializr или добавив следующий элемент `<dependency>` в файл `pom.xml` проекта:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Затем, чтобы не приходилось постоянно просматривать журналы сервера Admin в поисках случайно сгенерированного пароля, можно настроить учетную запись администратора в `application.yml`:

```
spring:
  security:
    user:
      name: admin
      password: 53cr3t
```

После этого, при попытке открыть веб-интерфейс сервера Admin в браузере, вам будет предложено ввести имя пользователя и пароль в форме входа Spring Security по умолчанию. Введите настроенные имя пользователя и пароль, чтобы войти.

По умолчанию Spring Security включает поддержку CSRF (Cross-Site Request Forgery – защита от межсайтовой подделки запросов) на сервере Spring Boot Admin, которая препятствует регистрации клиентских приложений на сервере. Поэтому необходимо добавить дополнительные настройки, чтобы отключить CSRF:

```
package tacos.admin;

import org.springframework.context.annotation.Bean;
import org.springframework.security.config.annotation.web.reactive
    .EnableWebFluxSecurity;
import org.springframework.security.config.web.server.ServerHttpSecurity;
import org.springframework.security.web.server.SecurityWebFilterChain;

@EnableWebFluxSecurity
public class SecurityConfig {

    @Bean
    public SecurityWebFilterChain filterChain(ServerHttpSecurity http) throws
        Exception {
        return http
            .csrf()
            .disable()
            .build();
    }
}
```

Конечно, эти настройки обеспечивают не особенно надежную защиту. Я советую вернуться к главе 5, где рассказывается о способах настройки Spring Security для обеспечения более надежной защиты сервера Admin.

16.3.2 Аутентификация в Actuator

В разделе 15.4 обсуждались вопросы защиты конечных точек Actuator с помощью базовой HTTP-аутентификации. Настроив эту защиту, можно воспрепятствовать доступу к конечным точкам Actuator тех, кто не знает имени пользователя и пароля, установленных вами. К сожалению, это означает, что сервер Admin тоже не сможет использовать конечные точки Actuator, если не передаст учетные данные. Но как сервер Admin может получить эти данные?

Если приложение регистрируется на сервере Admin непосредственно, то оно может передать свои учетные данные серверу в этот момент. Вам нужно лишь настроить несколько свойств, чтобы включить такую возможность.

Свойства `spring.boot.admin.client.username` и `spring.boot.admin.client.password` задают учетные данные, которые сервер Admin может использовать для доступа к конечным точкам Actuator приложе-

ния. Следующий фрагмент из *application.yml* показывает, как установить эти свойства:

```
spring:
  boot:
    admin:
      client:
        url: http://localhost:9090
        username: admin
        password: 53cr3t
```

Свойства `username` и `password` должны настраиваться в каждом приложении, которое регистрируется на сервере Admin. Указанные значения должны соответствовать имени пользователя и паролю, которые должны передаваться в заголовке `Authorization` для базовой аутентификации HTTP. В этом примере этим свойствам присвоены значения `admin` и `53cr3t`, соответствующие учетным данным, настроенным для доступа к конечным точкам Actuator.

Итоги

- Сервер Spring Boot Admin использует конечные точки Actuator из одного или нескольких приложений Spring Boot и отображает полученные данные в удобном веб-приложении.
- Приложения Spring Boot могут сами регистрироваться на сервере Admin, или сервер может автоматически обнаруживать их с помощью Eureka.
- В отличие от конечных точек Actuator, которые возвращают мгновенный снимок состояния приложения, сервер Admin способен отображать метрики, характеризующие работу приложения, в режиме реального времени.
- Сервер Admin упрощает фильтрацию результатов, возвращаемых конечными точками Actuator, и даже отображает некоторые данные в виде графиков.
- Сервер Admin – это приложение Spring Boot, поэтому его можно защитить любыми средствами, доступными в Spring Security.

17

Мониторинг Spring с помощью JMX

В этой главе рассматриваются следующие темы:

- работа с конечными точками Actuator MBean;
- отображение компонентов Spring в компоненты MBean;
- публикация уведомлений.

Уже более полутора десятилетий Java Management Extensions (JMX) считается стандартным средством мониторинга приложений на Java и управления ими. Экспортируя управляемые компоненты, известные как MBean (managed beans – управляемые компоненты), внешний клиент JMX может управлять приложением, вызывая операции, проверяя свойства и наблюдая за событиями в компонентах MBean.

В начале нашего знакомства с Spring и JMX мы посмотрим, как конечные точки Actuator отображаются в компоненты MBean.

17.1 Работа с компонентами MBean в Actuator

По умолчанию все конечные точки Actuator отображаются в компоненты MBean. Но начиная с версии Spring Boot 2.2 механизм JMX по умолчанию отключен. Чтобы включить его в приложении Spring Boot, можно присвоить свойству `spring.jmx.enabled` значение `true`. Вот как это выглядит в файле `application.yml`:

```
spring:
  jmx:
    enabled: true
```

Это свойство не только включает поддержку JMX в Spring, но также отображает все конечные Actuator в компоненты MBean. Для подключения к конечной точке Actuator MBean можно использовать любой JMX-клиент. Используя инструмент JConsole, входящий в комплект Java Development Kit, вы найдете MBean-компоненты Actuator в домене `org.springframework.boot`, как показано на рис. 17.1.

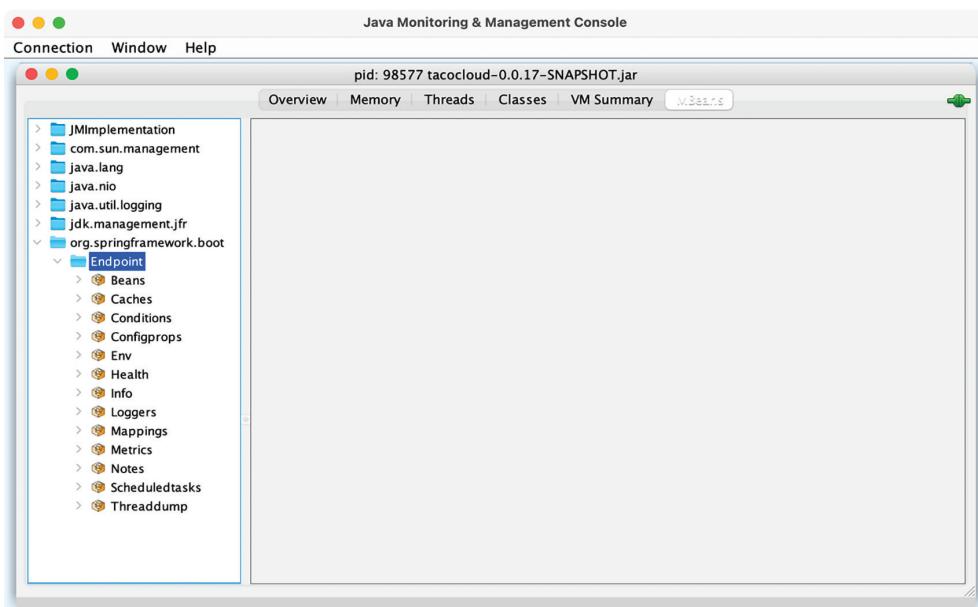


Рис. 17.1 Конечные точки Actuator автоматически отображаются в компоненты JMX MBean

Одно из замечательных свойств конечных точек Actuator MBean – все они доступны по умолчанию. Нет необходимости явно включать какую-либо из них, как это было с конечными точками HTTP. Однако есть возможность сузить выбор, установив свойства `management.endpoints.jmx.exposure.include` и `management.endpoints.jmx.exposure.exclude`. Например, чтобы сделать доступными только конечные точки Actuator MBean `/health`, `/info`, `/bean` и `/conditions`, нужно настроить свойство `management.endpoints.jmx.exposure.include`, как показано ниже:

```
management:
  endpoints:
    jmx:
      exposure:
        include: health,info,bean,conditions
```

Или, чтобы исключить несколько конечных точек, можно настроить свойство `management.endpoints.jmx.exposure.exclude`:

```
management:
  endpoints:
    jmx:
      exposure:
        exclude: env,metrics
```

Здесь с помощью свойства `management.endpoints.jmx.exposure.exclude` отключаются конечные точки `/env` и `/metrics`. Все остальные конечные точки Actuator по-прежнему будут отображаться в компоненты MBean.

Чтобы выполнить управляемую операцию в одном из компонентов MBean с помощью JConsole, распахните компонент MBean конечной точки в дереве слева, а затем выберите нужную операцию в разделе **Operations** (Операции).

Например, чтобы проверить уровни журналирования для пакета `tacos.ingredients`, распахните компонент MBean `Loggers` и выберите операцию `loggerLevels`, как показано на рис. 17.2. В форме вверху справа заполните поле **Name** (Имя), указав имя пакета (например, `org.springframework.web`), а затем щелкните на кнопке `loggerLevels`.

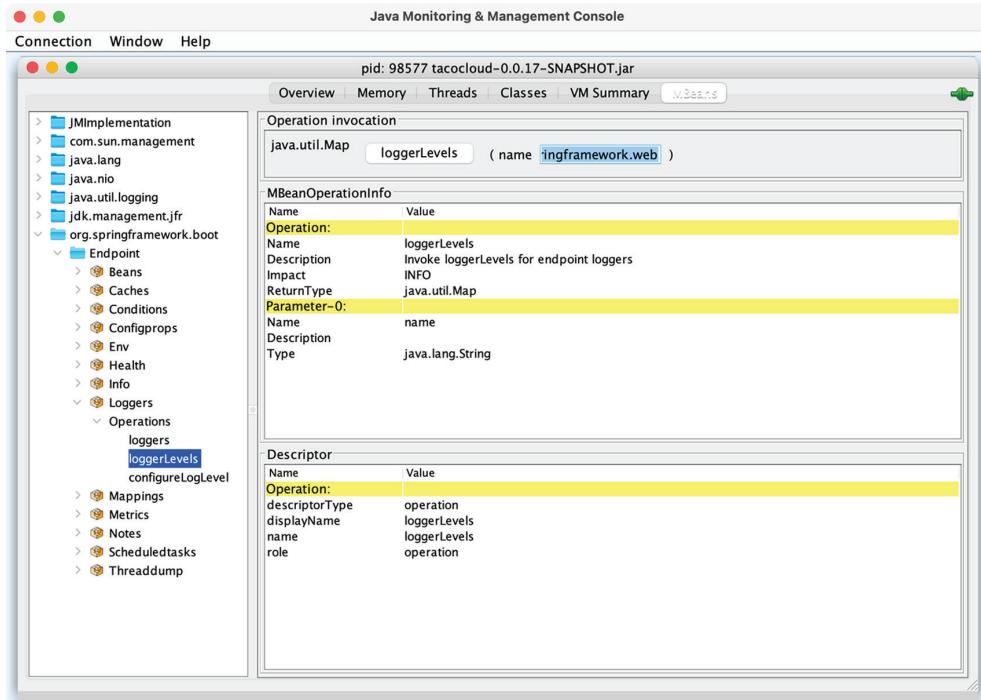


Рис. 17.2 Использование JConsole для просмотра уровней журналирования в приложении Spring Boot

После щелчка на кнопке `loggerLevels` появится диалог, показывающий ответ конечной точки MBean `/loggers`, как на рис. 17.3. Пользовательский интерфейс JConsole немного неудобен для работы, но вы быстро освоите его и научитесь использовать для просмотра содержимого любой конечной точки Actuator. Если вам не нравится JConsole, то ничего страшного – есть много других JMX-клиентов, из которых вы сможете выбрать тот, который больше придется вам по душе.

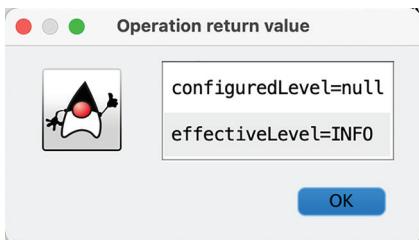


Рис. 17.3 Уровни журналирования, полученные из конечной точки MBean `/loggers` и отображаемые в JConsole

17.2 Создание своих компонентов MBean

Spring позволяет представить любой bean-компонент как компонент JMX MBean. Для этого нужно лишь снабдить класс компонента аннотацией `@ManagedResource`, а затем добавить аннотации `@ManagedOperation` и `@ManagedAttribute` к методам или свойствам соответственно. Об остальном позаботится Spring.

Например, предположим, что мы решили создать компонент MBean, предоставляющий информацию о количестве рецептов тако, созданных в Taco Cloud. Для этого можно определить компонент службы, ведущий подсчет количества рецептов тако. В листинге 17.1 показано, как может выглядеть такая служба.

Листинг 17.1 Компонент MBean, подсчитывающий количество рецептов тако

```
package tacos.jmx;

import java.util.concurrent.atomic.AtomicLong;
import org.springframework.data.rest.core.event.AbstractRepositoryEventListener;
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.stereotype.Service;
import tacos.Taco;
import tacos.data.TacoRepository;

@Service
@ManagedResource
public class TacoCounter
    extends AbstractRepositoryEventListener<Taco> {
```

```

private AtomicLong counter;

public TacoCounter(TacoRepository tacoRepo) {
    tacoRepo
        .count()
        .subscribe(initialCount -> {
            this.counter = new AtomicLong(initialCount);
        });
}

@Override
protected void onAfterCreate(Taco entity) {
    counter.incrementAndGet();
}

@ManagedAttribute
public long getTacoCount() {
    return counter.get();
}

@ManagedOperation
public long increment(long delta) {
    return counter.addAndGet(delta);
}

}

```

Класс `TacoCounter` снабжен аннотацией `@Service`, поэтому он будет обнаружен при сканировании компонентов, а его экземпляр – зарегистрирован как компонент в контексте приложения Spring. Но он также снабжен аннотацией `@ManagedResource`, указывающей, что этот компонент тоже должен отображаться в компонент MBean. Как компонент MBean он предоставляет один атрибут и одну операцию. Метод `getTacoCount()` отмечен аннотацией `@ManagedAttribute`, поэтому он будет отображаться как атрибут MBean, а метод `increment()` отмечен аннотацией `@ManagedOperation` и, соответственно, будет отображаться как операция MBean. На рис. 17.4 показано, как компонент MBean `TacoCounter` отображается в JConsole.

В классе `TacoCounter` кроется еще одна хитрость, хотя она не имеет ничего общего с JMX. Поскольку этот класс наследует `AbstractRepositoryEventLister`, он будет уведомляться о любых событиях сохранения экземпляров Taco в `TacoRepository`. В этом конкретном случае метод `onAfterCreate()` будет вызываться при создании и сохранении нового объекта Taco и увеличивать счетчик на единицу. Кроме того, `AbstractRepositoryEventLister` предлагает еще несколько методов для обработки событий до и после создания, сохранения или удаления объектов.

Работа с операциями и атрибутами MBean очень похожа на выполнение операций извлечения, т. е. если значение атрибута MBean изменится, вы не узнаете об этом, пока не прочтаете атрибут с помощью JMX-клиента. А теперь изменим ход игры и посмотрим, как организовать отправку уведомлений из MBean клиенту JMX.

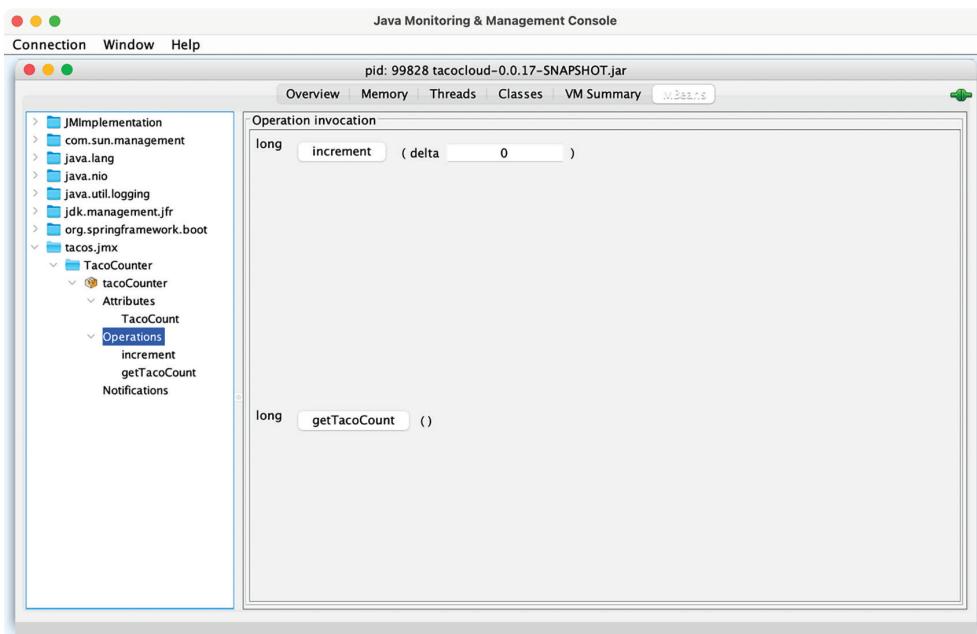


Рис 17.4 Операции и атрибуты TacoCounter в JConsole

17.3 Отправка уведомлений

Компоненты MBean могут отправлять уведомления клиентам JMX с помощью `Spring NotificationPublisher`. `NotificationPublisher` имеет единственный метод `sendNotification()`, который, получив объект `Notification`, публикует уведомление для всех клиентов JMX, подписавшихся на получение информации из компонента MBean.

Чтобы MBean мог публиковать уведомления, он должен реализовать интерфейс `NotificationPublisherAware`, который требует реализации метода `setNotificationPublisher()`. Например, предположим, что мы решили публиковать уведомление после каждого сотового созданного рецепта тако. Для этого можно изменить класс `TacoCounter`, реализовать в нем `NotificationPublisherAware` и использовать внедренный компонент `NotificationPublisher` для отправки уведомлений.

В листинге 17.2 показаны изменения в `TacoCounter`, которые необходимо внести для поддержки таких уведомлений.

Листинг 17.2 Отправка уведомления после создания каждого сотового рецепта тако

```
package tacos.jmx;

import java.util.concurrent.atomic.AtomicLong;
import org.springframework.data.rest.core.event.AbstractRepositoryEventListener;
```

```

import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedOperation;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.stereotype.Service;

import org.springframework.jmx.export.notification.NotificationPublisher;
import org.springframework.jmx.export.notification.NotificationPublisherAware;
import javax.management.Notification;

import tacos.Taco;
import tacos.data.TacoRepository;

@Service
@ManagedResource
public class TacoCounter
    extends AbstractRepositoryEventListener<Taco>
    implements NotificationPublisherAware {

    private AtomicLong counter;
    private NotificationPublisher np;

    @Override
    public void setNotificationPublisher(NotificationPublisher np) {
        this.np = np;
    }

    ...

    @ManagedOperation
    public long increment(long delta) {
        long before = counter.get();
        long after = counter.addAndGet(delta);
        if ((after / 100) > (before / 100)) {
            Notification notification = new Notification(
                "taco.count", this,
                before, after + "th taco created!");
            np.sendNotification(notification);
        }
        return after;
    }
}

```

Чтобы получать уведомления, в клиенте JMX необходимо подпітисься на MBean TacoCounter. Затем, после создания каждого сотого рецепта тако, клиент будет получать уведомления. На рис. 17.5 показано, как уведомления отображаются в JConsole.

Уведомления – это отличный способ активно отправлять данные и оповещать клиентов, не требующий от клиента опрашивать нужные атрибуты или вызывать операции.

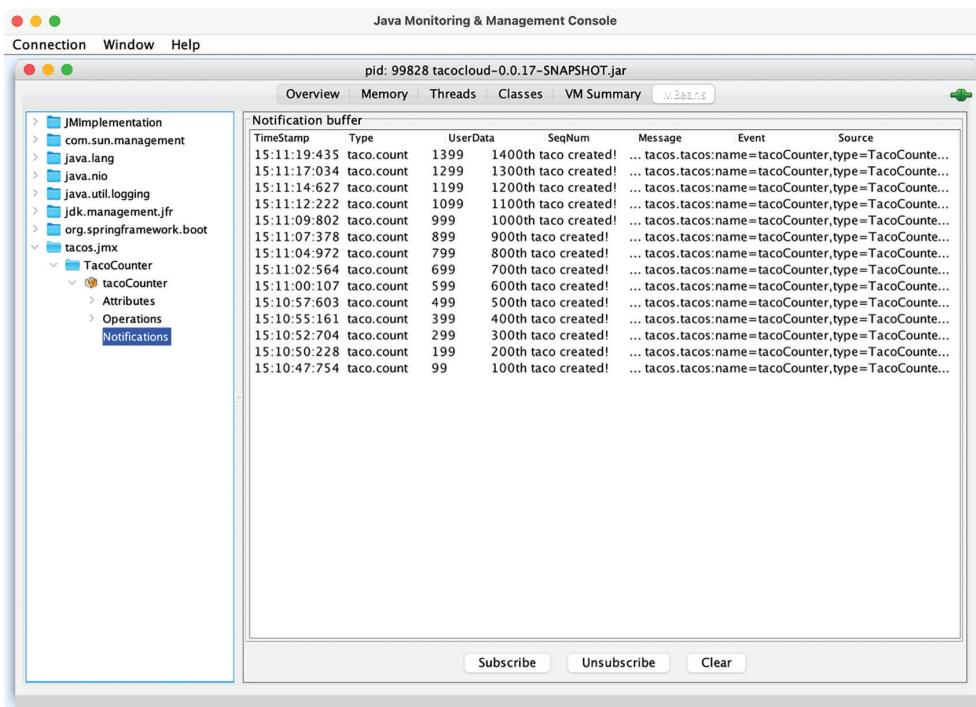


Рис. 17.5 JConsole, после подписки на MBean TacoCounter, получает уведомление о каждом сотом созданном рецепте тако

Итоги

- Большинство конечных точек Actuator доступны в виде компонентов MBean, к которым можно обращаться с помощью любого клиента JMX.
- Spring автоматически включает поддержку JMX для мониторинга компонентов в контексте приложения Spring.
- Компоненты Spring можно отобразить в компоненты MBean, снабдив их аннотацией `@ManagedResource`. Доступ к их методам и свойствам можно предоставить добавлением аннотаций `@ManagedOperation` и `@ManagedAttribute`.
- Компоненты Spring могут публиковать уведомления для передачи клиентам JMX с помощью `NotificationPublisher`.

18

Развертывание Spring

В этой главе рассматриваются следующие темы:

- сборка приложений Spring в виде файлов WAR или JAR;
- сборка приложений Spring в виде образов контейнеров;
- развертывание приложений Spring в Kubernetes.

Вспомните свой любимый боевик. А теперь представьте, что вы приходите в кинотеатр, наблюдаете захватывающее действие на экране с погонями, взрывами и стрельбой, и вдруг, как раз перед тем, как хорошие парни должны сразиться с плохими, фильм внезапно останавливается, в зале зажигается свет и всех выводят за дверь. Предыстория была захватывающей, но важна кульминация фильма. Без нее фильм получается незавершенным.

Теперь представьте, что вы разрабатываете приложение, вкладываете массу сил и энергии в решение бизнес-проблемы, но так и не развертываете его в помощь и на радость другим. Конечно, большинство приложений, которые мы пишем, не связаны с автомобильными погонями или взрывами (по крайней мере, я надеюсь на это), но в процессе нередко возникает определенная спешка. Не каждая строка кода, написанная вами, предназначена для использования в производстве, но было бы большим разочарованием, если бы ни одна из них так и не была бы развернута.

До настоящего момента все свое внимание мы уделяли использованию возможностей Spring Boot, помогающих разрабатывать при-

ложения. На этом пути мы узнали много нового и интересного, но все это будет напрасной тратой сил, если вы не пересечете финишную черту и не развернете приложение.

В этой главе мы оставим в стороне вопросы разработки приложений с помощью Spring Boot и обсудим порядок их развертывания. Любому, кто когда-либо развертывал приложения на Java, обсуждение этой темы может показаться ненужным, но фреймворк Spring Boot и связанные с ним проекты Spring предлагают дополнительные возможности, делающие развертывание приложений Spring Boot уникальным.

На самом деле, в отличие от большинства веб-приложений на Java, которые обычно развертываются на сервере приложений в виде файлов WAR, фреймворк Spring Boot предлагает несколько вариантов развертывания. Прежде чем перейти к обсуждению особенностей развертывания приложений Spring Boot, перечислим все доступные варианты и выберем те из них, которые лучше всего соответствуют нашим потребностям.

18.1 Варианты развертывания

Приложения Spring Boot можно создавать и запускать несколькими способами:

- непосредственно в интегрированной среде разработки (IDE) с помощью Spring Tool Suite или IntelliJ IDEA;
- из командной строки с помощью цели Maven `spring-boot: run` или задачи Gradle `bootRun`;
- с помощью Maven или Gradle создать выполняемый файл JAR, который можно запустить из командной строки или развернуть в облаке;
- с помощью Maven или Gradle создать файл WAR, который можно развернуть на традиционном сервере приложений Java;
- с помощью Maven или Gradle создать образ контейнера, который можно развернуть везде, где поддерживаются контейнеры, включая Kubernetes.

Любой из этих вариантов подходит для запуска приложения во время его разработки. Но какой вариант использовать для развертывания приложения в промышленном или другом окружении, не пред назначенном для разработки?

Вариант с запуском приложения из IDE или через Maven, или через Gradle не считается вариантом, пригодным для использования в промышленном окружении, а вот выполняемые файлы JAR и традиционные файлы WAR, напротив, прекрасно подходят для развертывания приложений. Теперь, имея три варианта развертывания – в виде файла WAR, файла JAR или образа контейнера, – какой выбрать? Обычно выбор зависит от того, где планируется развернуть приложение: на традиционном сервере приложений Java или на облачной платформе:

- *развертывание в облаке «платформа как услуга» (Platform as a Service, PaaS)*. Если планируется развернуть приложение на облачной платформе PaaS, такой как Cloud Foundry (<https://www.cloudfoundry.org/>), то лучшим выбором будет вариант с созданием выполняемого файла JAR. Даже если облачная платформа поддерживает развертывание файлов WAR, формат JAR намного проще и специально разрабатывался для развертывания на сервере приложений;
- *развертывание на серверах приложений Java*. Если планируется развернуть приложение на Tomcat, WebSphere, WebLogic или любом другом традиционном сервере приложений Java, то у вас действительно не останется другого выбора, кроме варианта с созданием файла WAR;
- *развертывание в Kubernetes*. Многие современные облачные платформы основаны на Kubernetes (<https://kubernetes.io/>). При развертывании в системе оркестрации контейнеров Kubernetes очевидным выбором является встраивание приложения в образ контейнера.

Учитывая вышесказанное, далее мы сосредоточимся на трех сценариях развертывания:

- сборка приложения Spring Boot в виде выполняемого файла JAR, который можно развернуть на платформе PaaS;
- развертывание приложения Spring Boot в виде файла WAR на сервере приложений Java, таком как Tomcat;
- упаковка приложения Spring Boot в виде образа контейнера Docker для развертывания на любой платформе, поддерживающей Docker.

Для начала рассмотрим самый, пожалуй, распространенный способ сборки приложений Spring Boot: в виде выполняемого файла JAR.

18.2 Сборка выполняемых файлов JAR

Сборка приложения Spring в выполняемый файл JAR производится довольно просто. Если предположить, что мы выбрали упаковку JAR при инициализации проекта, то мы сможем собрать выполняемый файл JAR с помощью следующей команды Maven:

```
$ mvnw package
```

Получившийся файл JAR будет помещен в каталог *target* и получит имя, сконструированное из значений, указанных в элементах *<artifactId>* и *<version>* в файле проекта *pom.xml* (например, *tacocloud-0.0.19-SNAPSHOT.jar*).

Если используется Gradle, то сборку можно запустить командой

```
$ gradlew build
```

При использовании системы сборки на основе Gradle получившийся файл JAR будет помещен в каталог *build/libs* и получит имя, основанное на значениях свойств *rootProject.name* в файле *setting.gradle* и *version* в *build.gradle*.

Имея выполнимый файл JAR, его можно запустить командой `java -jar`:

```
$ java -jar tacocloud-0.0.19-SNAPSHOT.jar
```

Приложение запустится и, если это веб-приложение, запустит встроенный сервер (Netty или Tomcat, в зависимости от того, является ли проект реактивным веб-проектом), после чего начнет принимать запросы на порте, указанном в свойстве *server.port* (по умолчанию 8080).

Этот вариант отлично подходит для запуска приложения на локальном компьютере. Но как развернуть выполнимый файл JAR?

Многое зависит от того, где будет развертываться приложение. Если развертывание выполняется в Cloud Foundry, то можно просто отправить файл JAR с помощью инструмента командной строки *cf*:

```
$ cf push tacocloud -p target/tacocloud-0.0.19-SNAPSHOT.jar
```

Первый аргумент *push* команды *cf* определяет имя, присвоенное приложению в Cloud Foundry. Это имя используется для ссылки на приложение в Cloud Foundry и интерфейсе командной строки *cf*, а также в качестве поддомена, в котором размещается приложение. Например, если домен вашего приложения в Cloud Foundry называется *cf.myorg.com*, то приложение Taco Cloud будет доступно по адресу <https://tacocloud.cf.myorg.com>.

Другой способ развернуть выполнимый файл JAR – упаковать его в контейнер Docker и запустить под управлением Docker или Kubernetes. Давайте посмотрим, как это сделать.

18.3 Сборка образа контейнера

Docker (<https://www.docker.com/>) стал фактическим стандартом распространения приложений всех видов для развертывания в облаке. Многие облачные платформы, включая AWS, Microsoft Azure и Google Cloud Platform (и это лишь некоторые из них), принимают контейнеры Docker для развертывания приложений.

Идея контейнерных приложений, таких как созданные с помощью Docker, аналогична транспортным контейнерам, которые используются для доставки товаров по всему миру. Все транспортные контейнеры имеют стандартный размер и формат независимо от их содержимого. Благодаря этому транспортные контейнеры легко штабелируются на кораблях, перевозятся поездами или грузовиками. Аналогично контейнерные приложения упаковываются в контейне-

ры общего формата, которые можно развертывать и запускать где угодно, независимо от приложений, находящихся внутри.

Самый простой способ создать образ приложения Spring Boot – определить файл Dockerfile и вызвать команду `docker build`, которая скопирует выполняемый файл JAR в образ контейнера. Следующий чрезвычайно простой файл Dockerfile определяет именно эти действия:

```
FROM openjdk:11.0.12-jre
ARG JAR_FILE=target/*.jar
COPY ${JARFILE} app.jar
ENTRYPOINT ["java","-jar","/app.jar"]
```

Файл Dockerfile описывает, как будет создаваться образ контейнера. Поскольку он такой короткий, рассмотрим его построчно:

- строка 1 объявляет, что создаваемый образ будет основан на предварительно определенном образе контейнера, который (среди прочего) содержит среду выполнения Open JDK 11 Java;
- строка 2 определяет переменную, ссылающуюся на все файлы JAR в каталоге `target/` проекта. В большинстве случаев должен быть только один файл JAR. Однако, используя подстановочный знак, можно указать в определении переменной в Dockerfile несколько файлов JAR. Путь к файлу JAR предполагает, что Dockerfile находится в корне проекта Maven;
- строка 3 копирует файл JAR из каталога `target/` проекта в образ контейнера с обобщенным именем `app.jar`;
- строка 4 задает точку входа, т. е. определяет команду запуска приложения при запуске контейнера, созданного из этого образа, в данном случае `java -jar /app.jar`.

Имея этот Dockerfile, можно создать образ с помощью инструмента командной строки Docker:

```
$ docker build . -t habuma/tacocloud:0.0.19-SNAPSHOT
```

Точка `(.)` в этой команде задает относительный путь к местоположению Dockerfile. Если команда `docker build` запускается из другого каталога, то замените точку `(.)` путем к каталогу с файлом Dockerfile (без имени файла). Например, если `docker build` запускается из родительского проекта, то команда должна выглядеть так:

```
$ docker build tacocloud -t habuma/tacocloud:0.0.19-SNAPSHOT
```

Значение аргумента `-t` – это тег образа, состоящий из имени и версии. В данном случае образ получит имя `habuma/tacocloud` и номер версии 0.0.19-SNAPSHOT. При желании можно сразу после сборки попробовать запустить контейнер командой `docker run`:

```
$ docker run -p8080:8080 habuma/tacocloud:0.0.19-SNAPSHOT
```

Параметр `-p8080:8080` обеспечивает пересылку запросов, поступающих в порт 8080 хоста (например, локального компьютера, где используется Docker), в порт 8080 контейнера (где запросы принимает Tomcat или Netty).

Создать образ Docker таким способом при наличии готового файла JAR достаточно просто, но создать тот же образ из исходного кода приложения Spring Boot сложнее. Начиная с версии Spring Boot 2.3.0 появилась возможность создавать образы контейнеров без добавления каких-либо специальных зависимостей, файлов конфигурации или включения какого-либо особого кода в проект, потому что плагины сборки Spring Boot для Maven и Gradle напрямую поддерживают создание образов контейнеров. Чтобы собрать проект Spring с помощью системы сборки Maven в образ контейнера, нужно задать цель `build-image` плагина Spring Boot Maven:

```
$ mvnw spring-boot:build-image
```

Если проект собирается с помощью системы сборки Gradle, то образ контейнера можно создать так:

```
$ gradlew bootBuildImage
```

Обе эти команды создадут образ контейнера с тегом по умолчанию, сконструированным на основе свойств `<artifactId>` и `<version>` в файле `pom.xml`. Для приложения Taco Cloud тег будет выглядеть как-то так: `library/tacocloud:0.0.19-SNAPSHOT`. Вскоре мы увидим, как указать свой тег для образа.

Плагины сборки Spring Boot полагаются на Docker. Поэтому на компьютер, где создается образ, необходимо установить среду выполнения Docker. Сразу после создания образа его можно запустить командой

```
$ docker run -p8080:8080 library/tacocloud:0.0.19-SNAPSHOT
```

Эта команда запускает образ и откроет порт 8080 образа (который прослушивает встроенный сервер Tomcat или Netty), в который будут пересыпаться запросы, поступающие в порт 8080 хоста.

Формат тега по умолчанию: `docker.io/library/${project.artifactId}:${project.version}`. Это объясняет, почему тег начинается со слова `library`. Это никак не мешает, если образ будет запускаться только локально. Но, скорее всего, вы захотите отправить свой образ в реестр образов, такой как DockerHub, а для этого необходимо, чтобы образ был создан с тегом, ссылающимся на имя вашего репозитория образов.

Например, предположим, что репозиторий вашей организации в DockerHub имеет имя `tacocloud`. В таком случае вам нужно, чтобы образ имел имя `tacocloud/tacocloud:0.0.19-SNAPSHOT`, то есть с префиксом `tacocloud` вместо `library`. Для этого нужно лишь задать свойство сборки. Для Maven имя образа задается с использованием

системного свойства JVM `spring-boot.build-image.imageName` следующим образом:

```
$ mvnw spring-boot:build-image \
-Dspring-boot.build-image.imageName=tacocloud/tacocloud:0.0.19-SNAPSHOT
```

При использовании системы сборки Gradle все еще проще. Имя образа можно задать в параметре `--imageName`:

```
$ gradlew bootBuildImage --imageName=tacocloud/tacocloud:0.0.19-SNAPSHOT
```

В обоих случаях вы должны помнить об этом, создавая образы, и не допускать ошибок. Чтобы упростить задачу, можно также указать имя образа как часть самой сборки.

В Maven имя образа можно указать в конфигурации плагина Spring Boot Maven. Например, следующий фрагмент из файла проекта `pom.xml` показывает, как задать имя образа в элементе `<configuration>`:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <configuration>
    <image>
      <name>tacocloud/${project.artifactId}:${project.version}</name>
    </image>
  </configuration>
</plugin>
```

Обратите внимание, что вместо жесткого определения идентификатора и версии артефакта можно использовать переменные сборки, значения которых ссылаются на определения где-то в другом месте. Это избавит от необходимости вручную увеличивать номер версии в имени образа по мере развития проекта. При использовании системы сборки Gradle данный эффект обеспечит следующая запись в `build.gradle`:

```
bootBuildImage {
  imageName = "habuma/${rootProject.name}:${version}"
}
```

С этими параметрами в спецификации сборки проекта можно создать образ командой, не содержащей имя образа. После сборки образ можно запустить командой `docker run`, как и раньше (сославшись на образ по его новому имени), или отправить образ в реестр, такой как DockerHub, командой `docker push`:

```
$ docker push habuma/tacocloud:0.0.19-SNAPSHOT
```

После отправки образа в реестр его можно извлечь и запустить в любом окружении, имеющем доступ к этому реестру. Для запуска образов все чаще используется Kubernetes, поэтому я предлагаю посмотреть, как запустить образ в Kubernetes.

18.3.1 Развёртывание в Kubernetes

Kubernetes – замечательная платформа оркестровки контейнеров, которая, кроме всего прочего, запускает образы, масштабирует контейнеры по мере необходимости и перезапускает аварийные контейнеры, повышая надежность.

Kubernetes – мощная платформа для развертывания приложений, обладающая настолько широкими возможностями, что у нас не получится подробно рассмотреть их все в этой главе. Поэтому мы сосредоточимся исключительно на задачах, связанных с развертыванием приложения Spring Boot, встроенного в образ контейнера, в кластере Kubernetes. Для более подробного знакомства с Kubernetes я рекомендую книгу «Kubernetes in Action, 2nd Edition» Марко Лукши (Marko Lukša)¹.

Система оркестровки контейнеров Kubernetes заработала репутацию сложной (возможно, несправедливо) в использовании. Однако в действительности развертывание приложений Spring, собранных в виде образов контейнеров, в системе Kubernetes выполняется просто и не лишено смысла, учитывая все преимущества, предоставляемые Kubernetes.

Для развертывания вам понадобится готовая система Kubernetes. Есть несколько вариантов, в том числе Amazon AWS EKS и Google Kubernetes Engine (GKE). Для локальных экспериментов можете запускать кластеры Kubernetes на локальных компьютерах, используя различные реализации Kubernetes, такие как MiniKube (<https://minikube.sigs.k8s.io/docs/>), MicroK8s (<https://microk8s.io/>) и моя любимая Kind (<https://kind.sigs.k8s.io/>).

Первое, что нужно сделать, – создать манифест развертывания. Манифест развертывания – это файл YAML, описывающий процесс развертывания образа. В качестве простого примера ниже приводится короткий манифест развертывания, развертывающий в кластере Kubernetes образ Taco Cloud, созданный ранее:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: taco-cloud-deploy
  labels:
    app: taco-cloud
spec:
  replicas: 3
  selector:
    matchLabels:
      app: taco-cloud
  template:
    metadata:
```

¹ Лукши Марко. Kubernetes в действии. ДМК-Пресс, 2018, ISBN: 978-5-97060-657-5. – Прим. перев.

```

labels:
  app: taco-cloud
spec:
  containers:
    - name: taco-cloud-container
      image: tacocloud/tacocloud:latest

```

Файл манифеста можно назвать как угодно. Но для обсуждения предположим, что мы назвали его *deploy.yaml* и поместили в каталог *k8s* в корне проекта.

Не вдаваясь в детали работы спецификации развертывания в Kubernetes, отмечу, что она называется *taco-cloud-deploy* и (внизу) настроена на развертывание и запуск контейнера на основе образа *tacocloud/tacocloud:latest*. Указав «*latest*» в качестве версии, вместо «*0.0.19-SNAPSHOT*», мы можем быть уверены, что будет использоваться самый последний образ, помещенный в реестр контейнеров.

Еще одна деталь, на которую следует обратить внимание, – свойство *replicas* со значением 3. Оно сообщает среде выполнения Kubernetes, что должно быть запущено три экземпляра контейнера. Если по какой-то причине один из этих трех экземпляров потерпит аварию, то Kubernetes автоматически решит проблему, остановив его и запустив новый экземпляр. Выполнить развертывание можно с помощью инструмента командной строки *kubectl*:

```
$ kubectl apply -f deploy.yaml
```

Спустя мгновение можно выполнить *kubectl get all*, чтобы увидеть результаты развертывания: в списке должно появиться три модуля (*pod*), в каждом из которых выполняется экземпляр контейнера. Вот что вы можете увидеть:

```

$ kubectl get all
NAME                               READY   STATUS    RESTARTS   AGE
pod/taco-cloud-deploy-555bd8fdb4-dln45  1/1    Running   0          20s
pod/taco-cloud-deploy-555bd8fdb4-n455b  1/1    Running   0          20s
pod/taco-cloud-deploy-555bd8fdb4-xp756  1/1    Running   0          20s

NAME                           READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/taco-cloud-deploy  3/3     3           3           20s

NAME                               DESIRED  CURRENT  READY   AGE
replicaset.apps/taco-cloud-deploy-555bd8fdb4  3        3         3        20s

```

В первом разделе присутствуют три модуля, по одному на каждый экземпляр, как мы и запросили в свойстве *replicas*. Раздел в середине – это сам ресурс развертывания. И последний раздел – специальный ресурс *ReplicaSet*, который Kubernetes использует, чтобы запомнить, сколько копий (реплик) приложения следует поддерживать.

Чтобы опробовать приложение, нужно открыть порт одного из модулей на вашем компьютере. Для этого выполните команду `kubectl port-forward`:

```
$ kubectl port-forward pod/taco-cloud-deploy-555bd8fdb4-dln45 8080:8080
```

В данном случае я выбрал первый из трех модулей, перечисленных в выводе команды `kubectl get all`, и попросил перенаправить запросы с порта 8080 хоста (на котором действует кластер Kubernetes) в порт 8080 модуля. После этого можно в браузере ввести адрес `http://localhost:8080` и соединиться с приложением Taco Cloud, выполняющимся в указанном модуле.

18.3.2 Корректное завершение работы

Есть несколько способов повысить совместимость приложений Spring с Kubernetes, но два самых важных – обеспечить корректное завершение работы и возможность проверки работоспособности.

В любой момент Kubernetes может решить остановить один или несколько модулей с нашим приложением. Это может быть обусловлено обнаружением проблемы или явным требованием остановить либо перезапустить модуль. Какой бы ни была причина, если приложение в модуле находится в процессе обработки запроса, то было бы неправильно немедленно завершить выполнение и оставить запрос необработанным, потому что в этом случае клиенту будет отправлено сообщение об ошибке, требующее повторить запрос.

Чтобы не обременять клиента сообщением об ошибке, можно реализовать корректное завершение работы в своем приложении Spring, просто присвоив свойству `server.shutdown` значение "graceful". Настроить это свойство можно в любом из источников, обсуждавшихся в главе 6, в том числе в `application.yml`:

```
server:  
  shutdown: graceful
```

Встретив такое требование корректного завершения работы, Spring будет откладывать завершение приложения на срок до 30 секунд, позволяя ему обработать любые незавершенные запросы. После того как все запросы будут обработаны или истечет время ожидания, приложению будет разрешено остановиться.

По умолчанию время ожидания составляет 30 секунд, но его можно переопределить, добавив свойство `spring.lifecycle.timeout-per-shutdown-phase`. Например, вот как можно уменьшить время ожидания до 20 секунд:

```
spring:  
  lifecycle.timeout-per-shutdown-phase: 20s
```

В период ожидания завершения работы встроенный сервер переходит принимать новые запросы. Это позволяет закончить обработку текущих запросов до того, как произойдет остановка.

Завершение работы – не единственный случай, когда приложение лишается возможности обрабатывать запросы. Например, во время запуска приложению может понадобиться некоторое время, чтобы подготовиться к обслуживанию трафика. Одним из способов, которым приложение Spring может сообщить Kubernetes, что оно пока не готово обслуживать трафик, является проверка готовности. Далее мы рассмотрим, как включить проверки работоспособности и готовности в приложении Spring.

18.3.3 Проверка готовности и жизнеспособности приложения

Как было показано в главе 15, конечная точка Actuator `/health` предоставляет признак работоспособности приложения. Но этот признак относится только к состоянию внешних зависимостей, используемых приложением, таких как базы данных или брокеры сообщений. Даже если приложение имеет возможность соединиться с базой данных, это не обязательно означает, что оно готово обрабатывать запросы или достаточно работоспособно, чтобы продолжать выполняться в своем текущем состоянии.

Kubernetes поддерживает понятие жизнеспособности и готовности к обслуживанию: индикаторы работоспособности приложения, помогающие системе Kubernetes определить, следует ли передавать трафик приложению или его лучше перезапустить, чтобы исправить возникшую проблему. Spring Boot поддерживает проверку работоспособности и готовности через конечную точку Actuator `/health` в виде подмножеств, известных как *группы работоспособности*.

Жизнеспособность – это показатель достаточной работоспособности приложения для продолжения работы без перезапуска. Если индикатор жизнеспособности приложения указывает, что оно неработоспособно, то система Kubernetes может отреагировать на это, завершив модуль с аварийным приложением и запустив новый.

Индикатор готовности, с другой стороны, сообщает системе Kubernetes, что приложение готово или не готово обслуживать трафик. Например, на этапе запуска приложению может потребоваться выполнить некоторые операции, прежде чем оно сможет начать обрабатывать запросы. В этот период индикатор готовности приложения может сообщать, что оно пока не готово, при этом приложение считается действующим нормально и Kubernetes не перезапустит его. Kubernetes будет учитывать состояние индикатора готовности, не отправляя запросов приложению. Как только приложение завершит инициализацию, оно может установить новое значение индикатора, и Kubernetes начнет передавать трафик приложению.

Включение проверок жизнеспособности и готовности

Чтобы включить проверки работоспособности и готовности в приложении Spring Boot, необходимо присвоить свойству `management.health.probes.enabled` значение `true`. Вот как это выглядит в файле `application.yml`:

```
management:  
  health:  
    probes:  
      enabled: true
```

После включения проверок ответ на запрос к конечной точке `/health` будет выглядеть примерно так (при условии что приложение полностью работоспособно):

```
{  
  "status": "UP",  
  "groups": [  
    "liveness",  
    "readiness"  
  ]  
}
```

Сама по себе базовая конечная точка `/health` мало что говорит нам о жизнеспособности или готовности приложения. Эту информацию можно получить, обратившись к конечным точкам `/actuator/health/liveness` и `/actuator/health/readiness`. В любом случае статус `UP` будет выглядеть так:

```
{  
  "status": "UP"  
}
```

Напротив, если приложение нежизнеспособно или не готово к обслуживанию трафика, то результат будет выглядеть так:

```
{  
  "status": "DOWN"  
}
```

В случае если конечная точка `readiness` приложения сообщает о неготовности к работе, то Kubernetes не будет направлять ему трафик. Если же конечная точка `liveness` сообщает о нежизнеспособности, то Kubernetes попытается исправить ситуацию, удалив модуль и запустив новый экземпляр.

Настройка проверки жизнеспособности и готовности в процессе развертывания

Теперь, когда мы настроили в Actuator конечные точки, сообщающие о жизнеспособности и готовности к работе, нужно сообщить Kubernetes о них в манифесте развертывания. В конце следующего манифеста развертывания показана конфигурация, сообщающая системе Kubernetes, как она может проверить состояния работоспособности и готовности:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: taco-cloud-deploy
  labels:
    app: taco-cloud
spec:
  replicas: 3
  selector:
    matchLabels:
      app: taco-cloud
  template:
    metadata:
      labels:
        app: taco-cloud
    spec:
      containers:
        - name: taco-cloud-container
          image: tacocloud/tacocloud:latest
          livenessProbe:
            initialDelaySeconds: 2
            periodSeconds: 5
            httpGet:
              path: /actuator/health/liveness
              port: 8080
          readinessProbe:
            initialDelaySeconds: 2
            periodSeconds: 5
            httpGet:
              path: /actuator/health/readiness
              port: 8080
```

Эта спецификация сообщает системе Kubernetes, что для получения признаков жизнеспособности и готовности к работе она должна отправить запрос GET по указанному пути в порт 8080. Согласно приведенным здесь настройкам, первый запрос должен выполняться через 2 секунды после запуска модуля приложения, а затем каждые 5 секунд.

Установка состояния жизнеспособности и готовности

Как устанавливаются состояния жизнеспособности и готовности? Эти состояния могут меняться внутренними механизмами самого фреймворка Spring или библиотеками, от которых зависит приложение. Для этого Spring или библиотеки могут публиковать событие изменения состояния доступности. Но это не единственная возможность; приложение тоже может содержать код, публикующий эти события.

Например, предположим, что мы решили отложить переход приложения в состояние полной готовности, пока не будет выполнена некоторая процедура инициализации. В начале жизненного цикла приложения, например в компоненте `ApplicationRunner` или `CommandLineRunner`, можно объявить о временной неготовности, чтобы отложить передачу трафика:

```
@Bean
public ApplicationRunner disableLiveness(ApplicationContext context) {
    return args -> {
        AvailabilityChangeEvent.publish(context,
            ReadinessState.REFUSING_TRAFFIC);
    };
}
```

Здесь `ApplicationRunner` получает экземпляр контекста приложения Spring в качестве параметра. Он необходим статическому методу `publish()` для публикации события. После завершения инициализации состояние готовности можно обновить, чтобы сообщить о готовности приложения к обслуживанию трафика:

```
AvailabilityChangeEvent.publish(context, ReadinessState.ACCEPTING_TRAFFIC);
```

Состояние жизнеспособности можно обновлять практически так же. Основное отличие заключается в именах событий: вместо `ReadinessState.ACCEPTING_TRAFFIC` и `ReadinessState.REFUSING_TRAFFIC` приложение должно публиковать события `LivenessState.CORRECT` и `LivenessState.BROKEN` соответственно. Например, если в процессе выполнения обнаружится фатальная ошибка, то приложение может запросить его перезапуск, опубликовав событие `Liveness.BROKEN`:

```
AvailabilityChangeEvent.publish(context, LivenessState.BROKEN);
```

Вскоре после этого Kubernetes прочитает состояние нежизнеспособности из конечной точки `liveness` и примет меры, перезапустив приложение. Этот период времени очень мал, но если приложение вдруг определило, что на самом деле оно может продолжить работу, то можно отменить событие `LivenessState.BROKEN`, опубликовав событие `LivenessState.CORRECT`:

```
AvailabilityChangeEvent.publish(context, LivenessState.CORRECT);
```

Пока система Kubernetes не прочитала конечную точку *liveness* после публикации события BROKEN, приложение считается работоспособным и могущим обслуживать трафик.

18.4 Создание и развертывание файлов WAR

На протяжении всей этой книги, когда вы разрабатывали код приложения Taco Cloud, мы запускали этот код в интегрированной среде разработки или из командной строки, как выполняемые файлы JAR. В любом случае встроенный сервер Tomcat (или Netty в приложениях Spring WebFlux) всегда был готов обслуживать запросы к приложению.

Во многом благодаря механизму автоконфигурации Spring Boot вы были избавлены от необходимости создавать файл *web.xml* или объявлять класс инициализации сервлета *DispatcherServlet* для Spring MVC. Но если предполагается развертывать приложение на сервере приложений Java, то необходимо создать файл WAR. И чтобы сервер приложений знал, как запускать приложение, необходимо также добавить в этот файл класс инициализации сервлета и объявить *DispatcherServlet*.

Как оказывается, создать файл WAR для приложения Spring Boot совсем несложно. На самом деле если вы выбрали вариант WAR при создании проекта приложения в Initializr, то больше ничего делать не нужно.

Приложение Initializr гарантирует, что сгенерированный проект будет содержать класс инициализатора сервлета, а файл сборки будет содержать инструкции для создания файла WAR. Однако если в Initializr вы выбрали сборку файла JAR (или если вам интересно увидеть различия), тогда читайте дальше.

Во-первых, вам понадобится возможность настроить *DispatcherServlet*. Это можно сделать с помощью файла *web.xml*, но Spring Boot предлагает более простой вариант: *SpringBootServletInitializer*. *SpringBootServletInitializer* – специальная реализация *WebApplicationInitializer*. Помимо настройки *DispatcherServlet*, компонент *SpringBootServletInitializer* также отыскивает в контексте приложения Spring любые компоненты типа *Filter*, *Servlet* или *ServletContextInitializer* и добавляет их в контейнер сервлета.

Чтобы воспользоваться *SpringBootServletInitializer*, нужно создать подкласс и переопределить метод *configure()*, дабы указать класс конфигурации Spring. В листинге 18.1 показан *TacoCloudServletInitializer* – подкласс *SpringBootServletInitializer*, который может использовать приложение Taco Cloud.

Листинг 18.1 Инициализация веб-приложения средствами Java

```
package tacos;

import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.context.web.SpringBootServletInitializer;
```

```
public class TacoCloudServletInitializer
    extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(
        SpringApplicationBuilder builder) {
        return builder.sources(TacoCloudApplication.class);
    }
}
```

Как видите, метод `configure()` получает компонент `SpringApplicationBuilder` в параметре и возвращает его в результате, вызывая перед этим метод `sources()`, который регистрирует конфигурационные классы Spring. В данном случае он регистрирует только класс `TacoCloudApplication`, который выполняет функцию класса начальной загрузки (для выполняемых файлов JAR) и конфигурационного класса Spring.

Несмотря на то что приложение имеет другие конфигурационные классы, нет необходимости регистрировать их все с помощью метода `sources()`. Класс `TacoCloudApplication` с аннотацией `@SpringBootApplication` неявно запускает механизм сканирования компонентов, который обнаруживает и извлекает любые другие найденные конфигурационные классы.

По большей части класс, наследующий `SpringBootServletInitializer`, является шаблонным. Он ссылается на основной конфигурационный класс приложения. Но в остальном он одинаков во всех приложениях, которые вы будете упаковывать в файл WAR, и вам едва ли придется вносить в него какие-либо изменения.

Теперь, определив класс инициализатора сервлета, нужно внести несколько небольших изменений в спецификацию сборки проекта. При использовании системы сборки Maven достаточно изменить элемент `<packaging>` в `pom.xml`:

```
<packaging>war</packaging>
```

Изменения в спецификации сборки при применении Gradle также просты – нужно лишь применить плагин `war` в файле `build.gradle`:

```
apply plugin: 'war'
```

Теперь можно приступать к сборке приложения. В системе сборки Maven для этого следует использовать сценарий-обертку, который `Initializr` применяет для выполнения цели `package`, например:

```
$ mvnw package
```

Если сборка выполнится успешно, то вы найдете файл WAR в каталоге `target`. Если сборка выполняется с помощью системы Gradle, то тогда следует использовать оболочку Gradle:

```
$ gradlew build
```

По завершении сборки файл WAR будет находиться в каталоге `build/libs`. Теперь осталось лишь развернуть приложение. Процедура развертывания зависит от сервера приложений, поэтому, чтобы узнать, как осуществляется развертывание, обращайтесь к документации для конкретного сервера приложений.

Интересно отметить, что только что созданный файл WAR, пригодный для развертывания в любом контейнере сервлетов Servlet 3.0 (или выше), по-прежнему можно запустить из командной строки, как если бы он был выполняемым файлом JAR:

```
$ java -jar target/taco-cloud-0.0.19-SNAPSHOT.war
```

Фактически вы получаете два варианта развертывания на основе одного артефакта!

18.5 Закончим тем, с чего начинали

На протяжении нескольких сотен страниц мы перешли от создания проекта приложения с помощью [start.spring.io](#) до его развертывания в облаке. Я надеюсь, что вы получили такое же удовольствие от чтения этих страниц, какое получил я, когда писал их.

Эта книга подошла к концу, но ваши приключения с фреймворком Spring только начинаются. Используя все, что вы узнали на этих страницах, вы сможете создавать потрясающие приложения с помощью Spring. И я с большим удовольствием познакомлюсь с вашими творениями!

Итоги

- Приложения Spring можно развертывать в разных окружениях, включая традиционные серверы приложений и окружения PaaS, такие как Cloud Foundry, а также в виде контейнеров Docker.
- Сборка в виде исполняемого файла JAR дает возможность развернуть приложение Spring Boot на нескольких облачных платформах без лишних хлопот, связанных с созданием файла WAR.
- При создании файла WAR нужно добавить класс, наследующий `SpringBootServletInitializer`, чтобы обеспечить правильную настройку `DispatcherServlet`.
- Чтобы создать образ приложения Spring в форме контейнера, достаточно лишь использовать плагин Spring Boot, поддерживающий создание образов. Затем полученные образы можно развернуть везде, где поддерживается развертывание контейнеров Docker, в том числе в кластерах Kubernetes.

Приложение A

Создание проектов приложений Spring

Проекты Spring можно создавать разными способами, и выбор конкретного способа во многом зависит от вашего личного вкуса. Многие варианты зависят от выбора интегрированной среды разработки (IDE), которую вы используете.

Все эти варианты, кроме одного, основаны на применении приложения Spring Initializr, которое предоставляет REST API для создания проекта приложения Spring Boot. Различные виды IDE – это не что иное, как клиенты этого REST API. Однако есть несколько способов использовать Spring Initializr API вне среды разработки, и мы кратко рассмотрим их в этом приложении.

A.1 Инициализация проекта с помощью Spring Tool Suite

Чтобы инициализировать новый проект Spring с помощью Spring Tool Suite, выберите в меню пункт **File > New > Spring Starter Project** (Файл > Создать > Начальный проект Spring), как показано на рис. А.1.

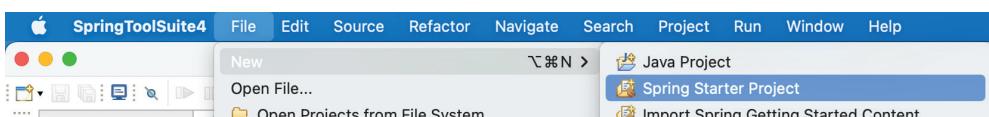


Рис. А.1 Создание нового проекта с помощью Spring Tool Suite

ПРИМЕЧАНИЕ Это лишь краткое описание использования Spring Tool Suite для инициализации проекта Spring. Подробнее об этом рассказывается в разделе 1.2.1.

На экране появится первая страница диалога создания проекта (рис. А.2). На этой странице введите основную информацию о проекте: имя проекта, идентификаторы группы и артефакта, версию и имя базового пакета. Также можно выбрать систему сборки проекта – Maven или Gradle, тип собираемого файла – JAR или WAR, версию Java и даже альтернативный язык, поддерживаемый JVM, например Groovy или Kotlin.

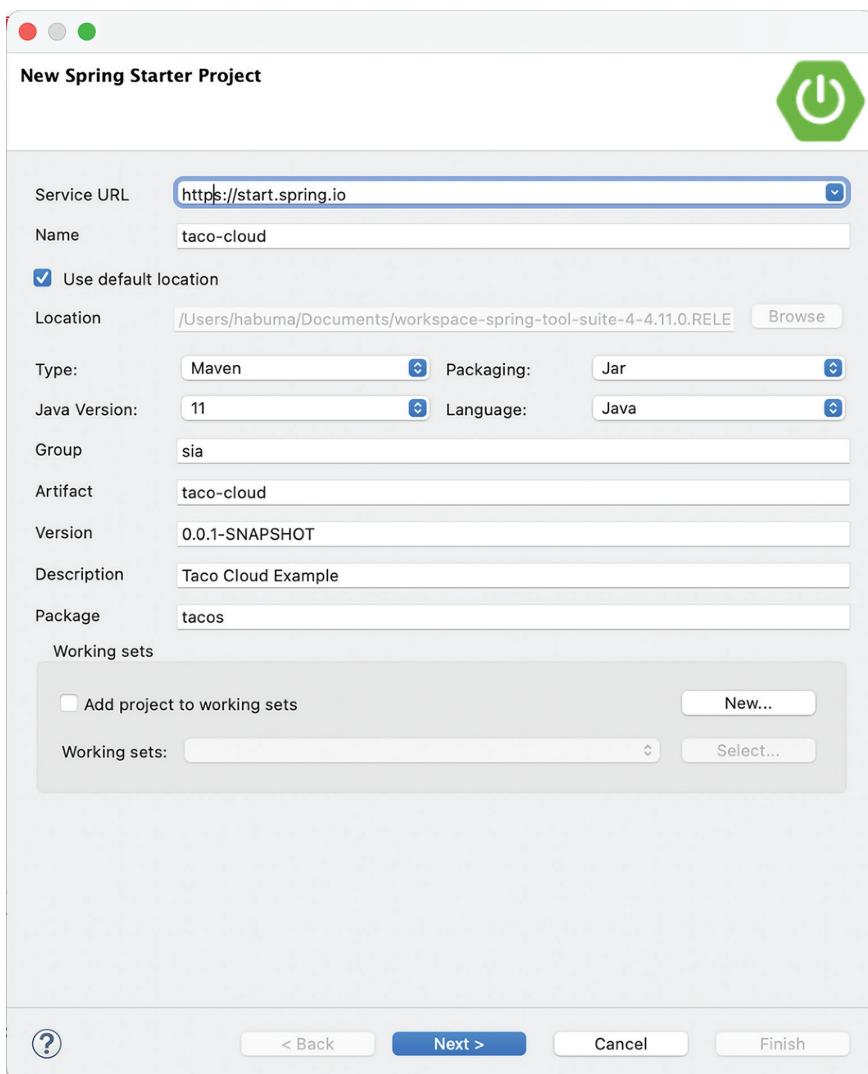


Рис. А.2 Определение основной информации о проекте

В первом поле на этой странице предлагается указать местоположение службы Spring Initializr. Если вы используете свой экземпляр Initializr, то укажите базовый URL службы Initializr. Иначе оставьте значение по умолчанию: <http://start.spring.io>.

Определив основную информацию о проекте, щелкните на кнопке **Next** (Далее), чтобы перейти на страницу зависимостей проекта (рис. А.3).

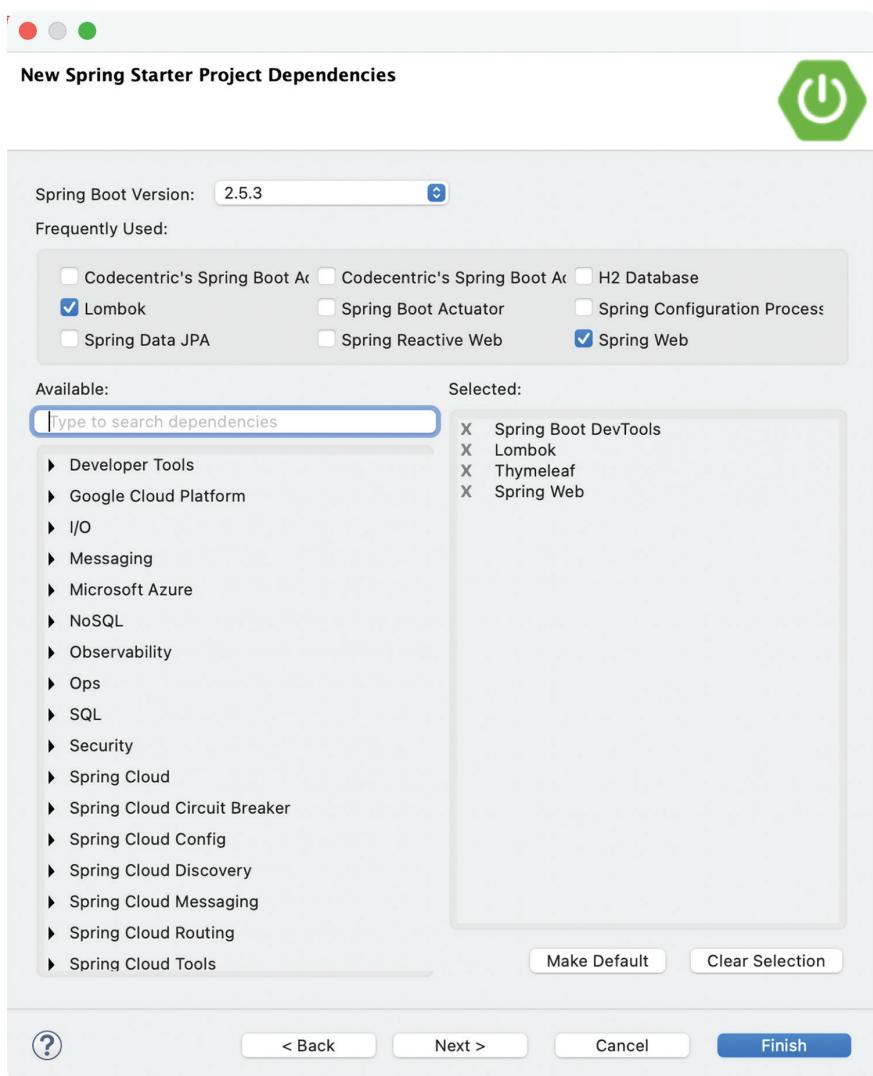


Рис. А.3 Зависимости проекта

На странице зависимостей проекта можно указать все зависимости, которые потребуются проекту. Многие из этих зависимостей яв-

ляются начальными зависимостями Spring Boot, но среди них есть и другие зависимости, которые часто используются в проектах Spring.

Доступные зависимости перечислены слева и организованы в группы, которые можно разворачивать или сворачивать. Если у вас возникнут проблемы с поиском зависимостей, то воспользуйтесь полем **Available** (Доступно), чтобы найти нужную зависимость.

Чтобы добавить зависимость в генерированный проект, установите флажок рядом с именем зависимости. Выбранная зависимость появится в списке справа под заголовком **Selected** (Выбрано). Выбранные зависимости можно удалить, щелкнув на крестике X рядом с выбранной зависимостью или на кнопке **Clear Selection** (Очистить выбор), чтобы удалить все выбранные зависимости.

Как дополнительное удобство, если вы часто используете в своих проектах определенный базовый набор зависимостей, можно после выбора таких зависимостей щелкнуть на кнопке **Make Default** (Сделать стандартными), и они всегда будут выбираться автоматически при создании каждого последующего проекта.

Выбрав зависимости, щелкните на кнопке **Finish** (Готово), чтобы генерировать проект и добавить его в рабочее пространство. Однако если вы используете Initializr, отличный от <http://start.spring.io>, то щелкните на кнопке **Next** (Далее), чтобы задать базовый URL Initializr, как показано на рис. А.4.

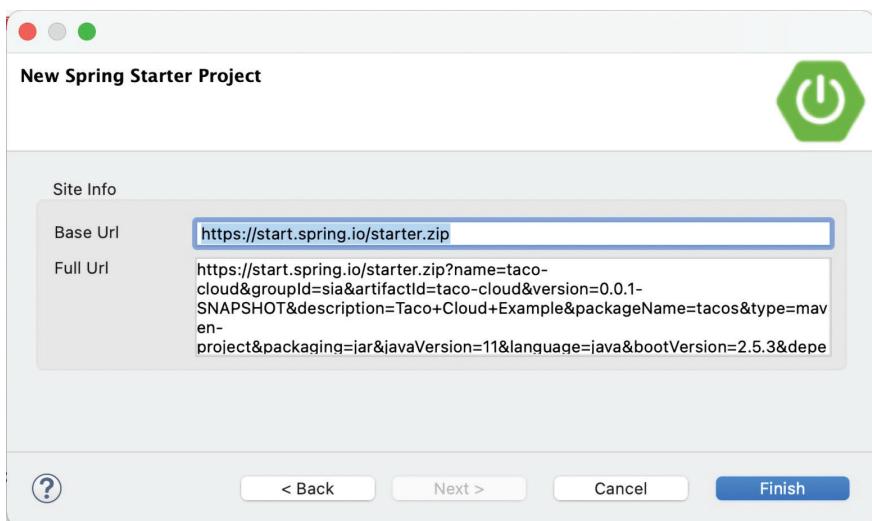


Рис. А.4 Выбор другого URL приложения Initializr

Поле **Base Url** (Базовый URL) содержит URL, который прослушивает Initializr API. Это единственное поле, которое можно изменить на данной странице. В поле **Full URL** (Полный URL) отображается полный URL, который будет использоваться для отправки приложению Initializr запроса на создание нового проекта.

A.2 Инициализация проекта с помощью IntelliJ IDEA

Чтобы создать новый проект Spring в IntelliJ IDEA, выберите в меню пункт **File > New > Project** (Файл > Создать > Проект), как показано на рис. A.5.

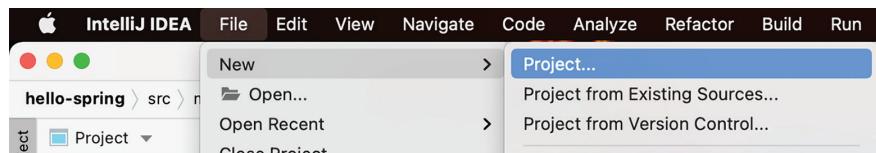


Рис. A.5 Создание нового проекта Spring в IntelliJ IDEA

На экране появится первая страница мастера создания проекта Spring Initializr. На этой странице вы должны ввести основную информацию о проекте, как показано на рис. A.6.

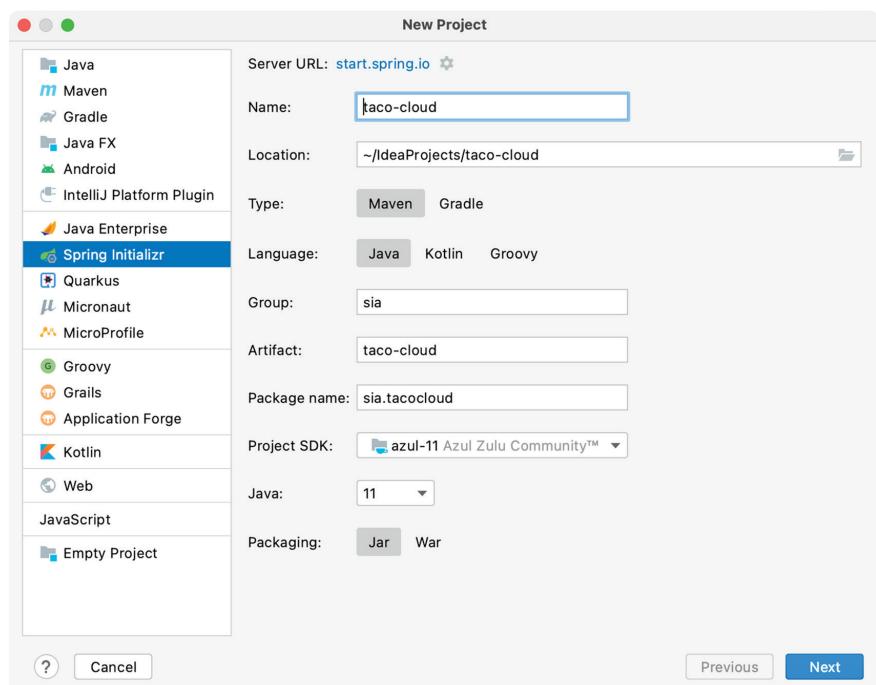


Рис. A.6 Определение основной информации о проекте в IntelliJ IDEA

Здесь вы без труда узнаете некоторые поля, соответствующие записям в файле Maven *pom.xml*, – на самом деле если вы выберете **Maven** в поле **Type** (Тип), то увидите именно эти поля. При желании може-

те выбрать тип **Gradle**, если предпочтаете пользоваться системой сборки Gradle.

После ввода основной информации о проекте щелкните на кнопке **Next** (Далее), чтобы отобразить страницу зависимостей проекта (рис. А.7).

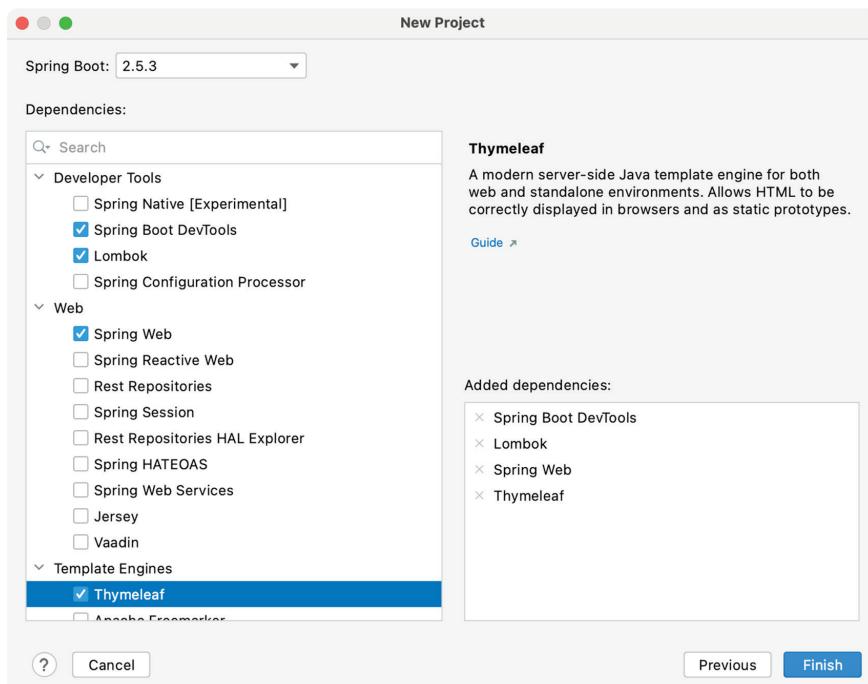


Рис. А.7 Зависимости проекта

Доступные зависимости перечислены слева и организованы в группы, которые можно разворачивать или сворачивать. Выбранные зависимости будут появляться (в соответствии с категорией) в списке справа.

Выбрав зависимости, щелкните на кнопке **Finish** (Готово), чтобы сгенерировать проект и добавить его в рабочее пространство IntelliJ IDEA.

A.3 Инициализация проекта с помощью NetBeans

Прежде чем создать новый проект Spring Boot в NetBeans, установите плагин поддержки Spring Boot в NetBeans. Плагин «NB Spring Boot» добавляет в NetBeans функции, аналогичные встроенным в Spring Tool Suite и IntelliJ IDEA.

Чтобы установить плагин, выберите в меню пункт **Tools > Plugins** (Инструменты > Плагины), как показано на рис. А.8.



Рис. А.8 Пункт меню Tools > Plugins (Инструменты > Плагины) в NetBeans

На экране появится список доступных плагинов для NetBeans, включая плагин «NB Spring Boot», как показано на рис. А.9.

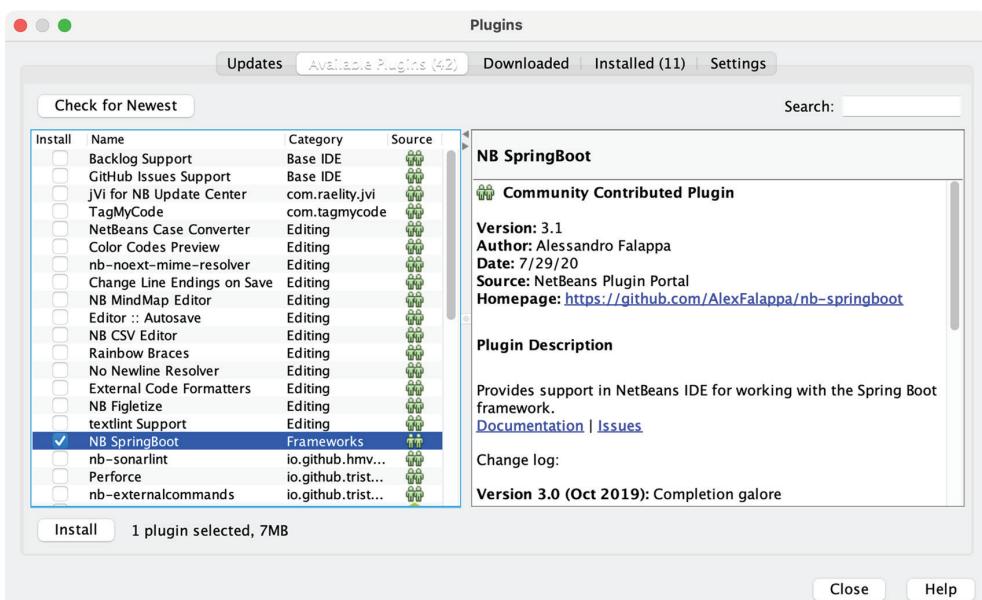


Рис. А.9 Выбор плагина «NB Spring Boot» для установки

Щелкните на кнопке **Install** (Установить), чтобы запустить установку плагина «NB Spring Boot». После этого вам будет предложено подтвердить свое решение и согласие с лицензионным соглашением плагина. Просто щелкайте на кнопке **Next** (Далее), пока не дойдете до последнего диалога, а затем щелкните на кнопке **Install** (Установить). По окончании установки вам будет предложено перезапустить NetBeans, чтобы задействовать плагин.

После установки плагина «NB Spring Boot» можно приступить к созданию нового проекта Spring Boot в NetBeans. Чтобы создать новый проект Spring в NetBeans, выберите в меню пункт **File > New Project** (Файл > Создать проект), как показано на рис. А.10.

На экране появится первая страница мастера создания нового проекта. Как показано на рис. А.11, на этой странице вы сможете выбрать тип создаваемого проекта.

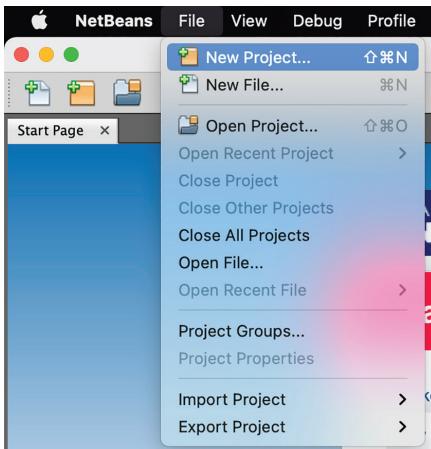


Рис. А.10 Создание нового проекта Spring в NetBeans

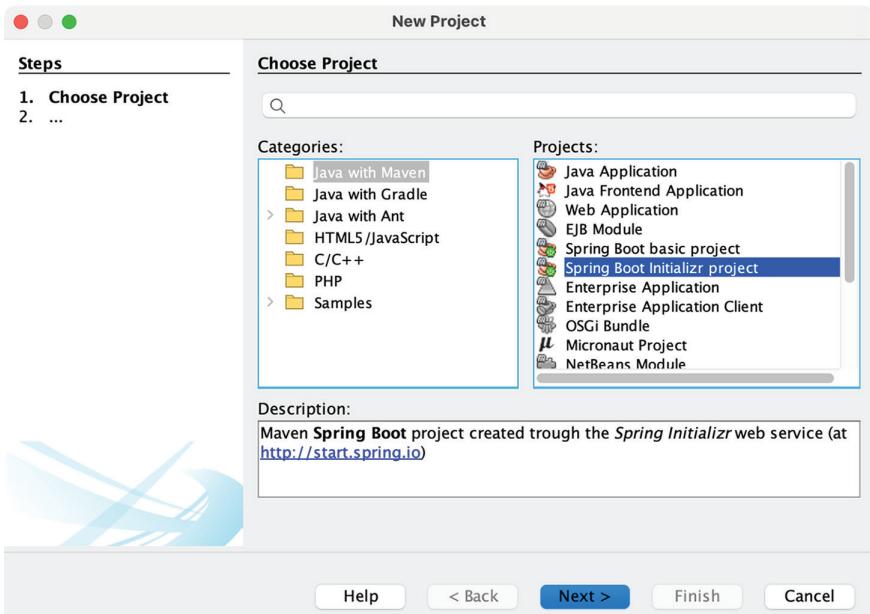


Рис. А.11 Создание нового проекта Spring Boot Initializr

Для проекта Spring Boot выберите **Java with Maven** (Java и Maven) в списке слева, а затем **Spring Boot Initializr Project** (Проект Spring Boot Initializr) в списке справа. После этого щелкните на кнопке **Next >** (Далее), чтобы перейти на следующую страницу.

На второй странице мастера (рис. А.12) введите основную информацию о проекте: имя проекта, версию и другие сведения, которые в конечном итоге будут использоваться для определения проекта в файле Maven *pom.xml*.

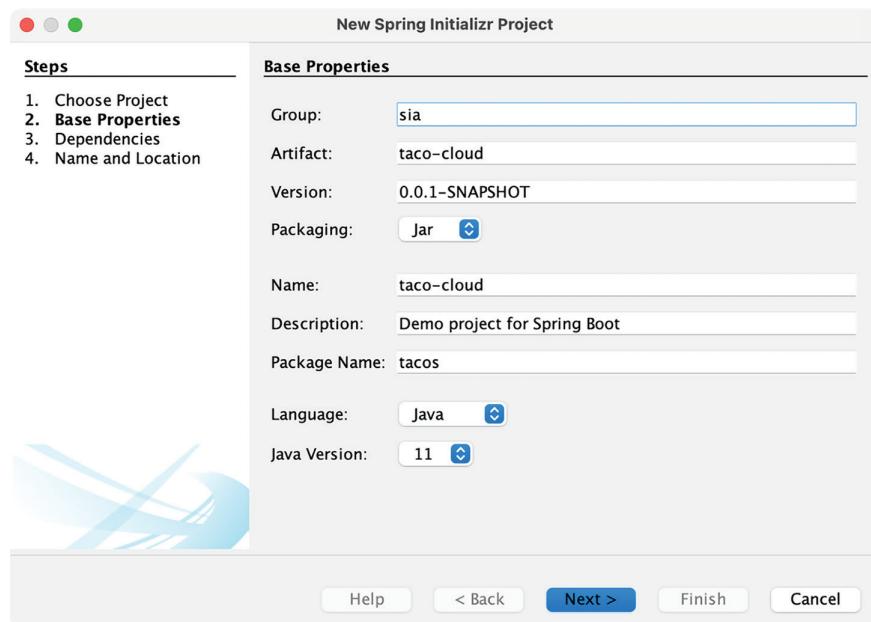


Рис. А.12 Определение основной информации о проекте

После ввода основной информации о проекте щелкните на кнопке **Next** (Далее), чтобы перейти на страницу зависимостей, как показано на рис. А.13.

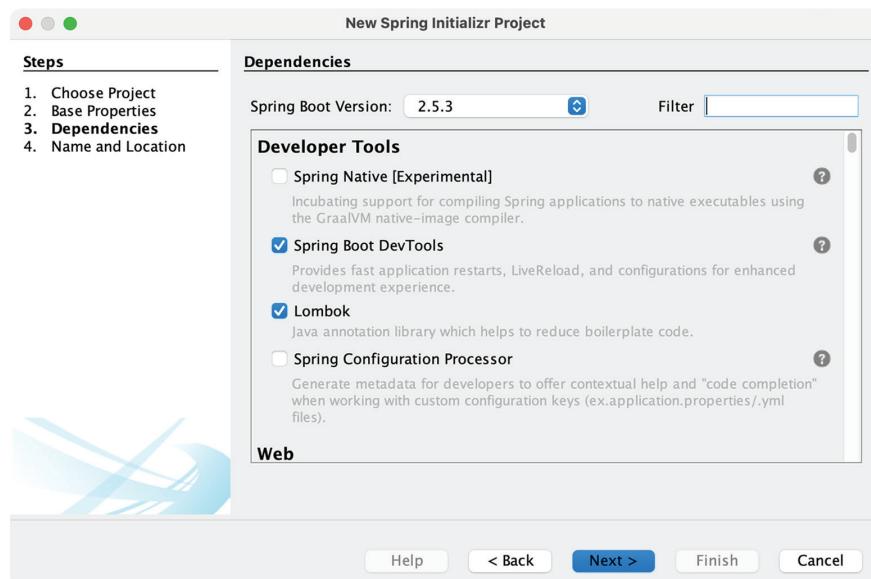


Рис. А.13 Зависимости проекта

Все зависимости перечислены в виде флагков в одном списке и упорядочены по категориям. Если у вас возникнут проблемы с поиском конкретной зависимости, используйте текстовое поле **Filter** (Фильтр) вверху, чтобы ограничить список параметров.

Здесь также можно указать версию Spring Boot для использования. По умолчанию будет установлена текущая общедоступная версия Spring Boot.

Выбрав зависимости, щелкните на кнопке **Next** (Далее), чтобы перейти к последней странице мастера, изображенной на рис. А.14. На этой странице можно указать некоторые дополнительные сведения о проекте, включая имя проекта и его местоположение в файловой системе. (Поле **Project Folder** (Папка проекта) доступно только для чтения и является производным от двух других полей.) Здесь также имеется флагок, управляющий возможностью запускать и отлаживать проект с помощью плагина Maven Spring Boot вместо NetBeans. Здесь же можно установить флагок, требующий от NetBeans удалить сценарий-обертку Maven из созданного проекта.

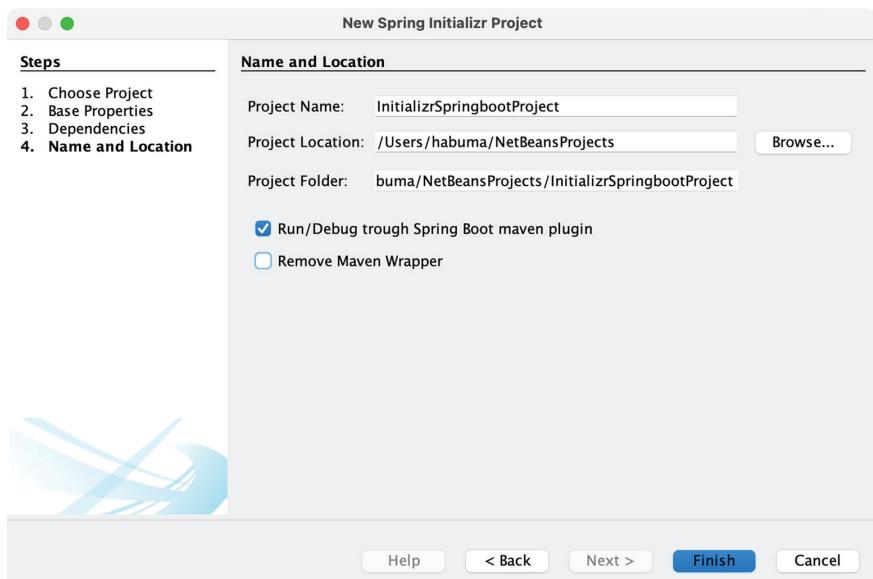


Рис. А.14 Настройка имени проекта и его местоположения

После ввода всей информации щелкните на кнопке **Finish** (Готово), чтобы сгенерировать проект и добавить его в рабочее пространство NetBeans.

A.4 Инициализация проекта с помощью start.spring.io

Скорее всего, один из описанных выше вариантов инициализации проекта в IDE удовлетворит потребности большинства, но некоторые из вас могут использовать совершенно другую IDE или предпочитают простой текстовый редактор. В этом случае все равно можно воспользоваться преимуществами Spring Initializr, используя веб-интерфейс этого приложения.

Для начала откройте в веб-браузере страницу <https://start.spring.io>. Вы увидите простую версию пользовательского веб-интерфейса Spring Initializr, изображенную на рис. А.15.

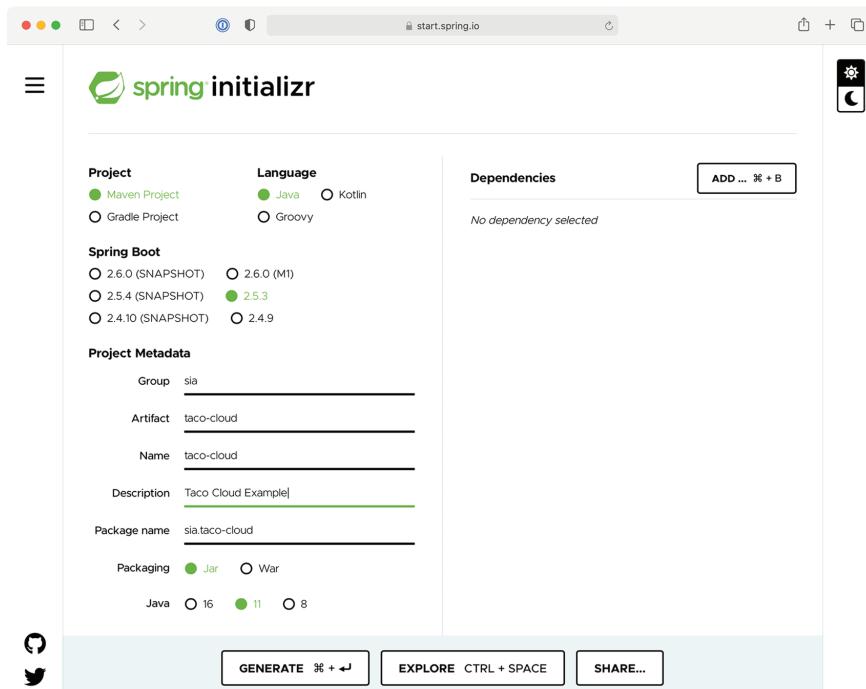


Рис. А.15 Простой веб-интерфейс приложения Spring Initializr

В этом веб-интерфейсе вы должны ввести некоторую основную информацию, в том числе выбрать систему сборки Maven или Gradle, язык программирования для проекта, версию Spring Boot, а также идентификаторы группы и артефакта проекта.

Здесь же у вас будет возможность настроить зависимости, используя поле поиска. Например, как показано на рис. А.16, можно ввести слово «web», чтобы отыскать все зависимости, содержащие ключевое слово «web».

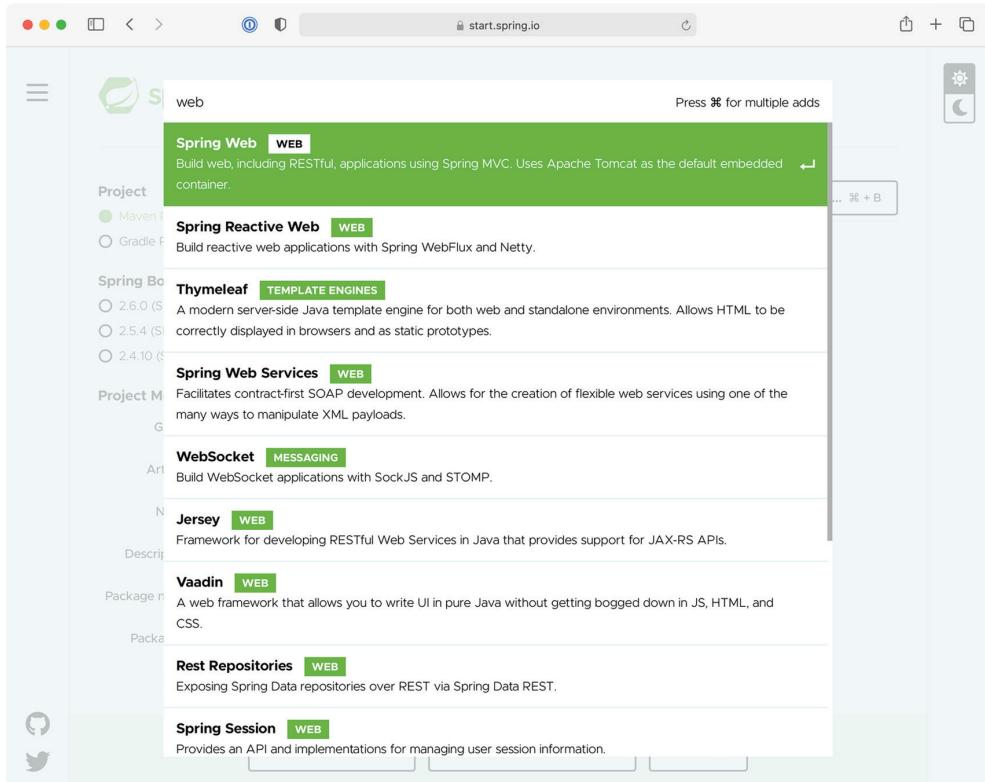


Рис. А.16 Поиск зависимостей проекта

Увидев нужную зависимость, нажмите клавишу **Enter** на клавиатуре, чтобы выбрать ее, и она будет добавлена в список выбранных зависимостей. Поле под заголовком **Selected Dependencies** (Выбранные зависимости) на рис. А.17 показывает, что были выбраны зависимости **Web**, **Thymeleaf**, **DevTools** и **Lombok**.

Если вы решите, что какая-то из выбранных зависимостей вам не нужна, то щелкните мышкой на значке **X** справа от названия зависимости, чтобы удалить ее. Закончив, щелкните на кнопке **Generate Project** (Создать проект) или нажмите комбинацию клавиш, указанную в надписи на кнопке (зависит от операционной системы), чтобы приложение Initializr сгенерировало проект и загрузило его в виде zip-файла. Этот файл затем нужно распаковать и открыть проект в любой IDE или редакторе по вашему выбору.

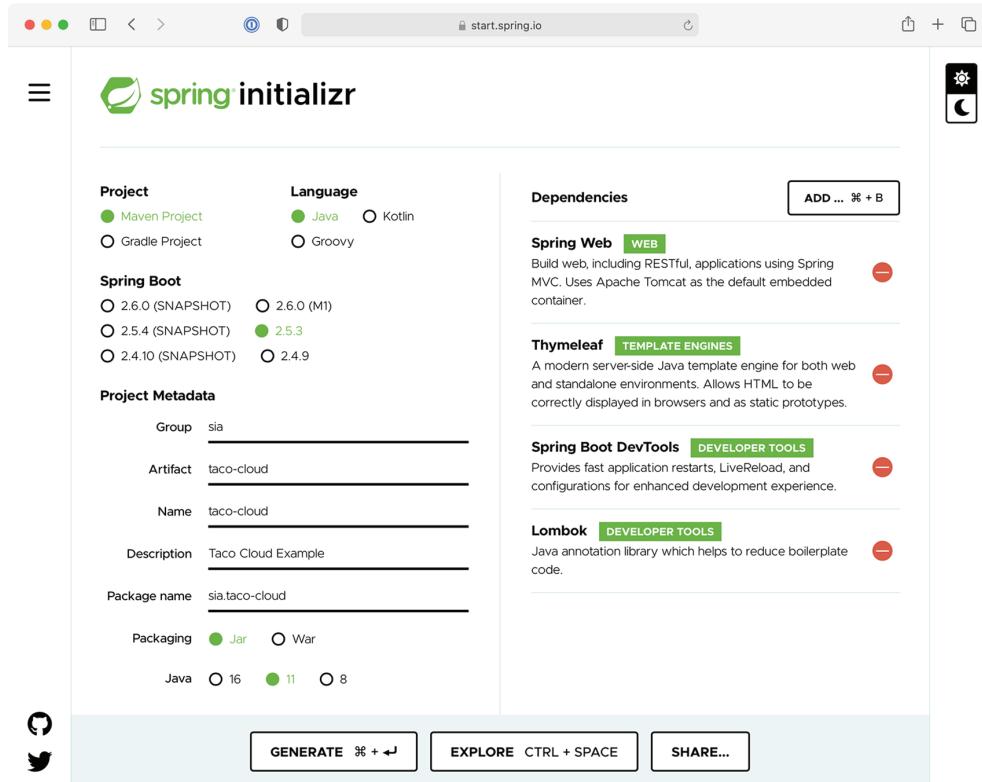


Рис. A.17 Выбор зависимостей проекта

Прежде чем щелкнуть на кнопке **Generate Project** (Создать проект), выполните обзор получившегося проекта, щелкнув на кнопке **Explore** (Исследовать). Появится диалог с проводником, как показано на рис. А.18.

Сначала будет показана спецификация сборки проекта (содержимое файла Maven *pom.xml* или файла Gradle *build.gradle*). Выбирая элементы в дереве слева, можно увидеть, какие еще артефакты будут включены в проект.

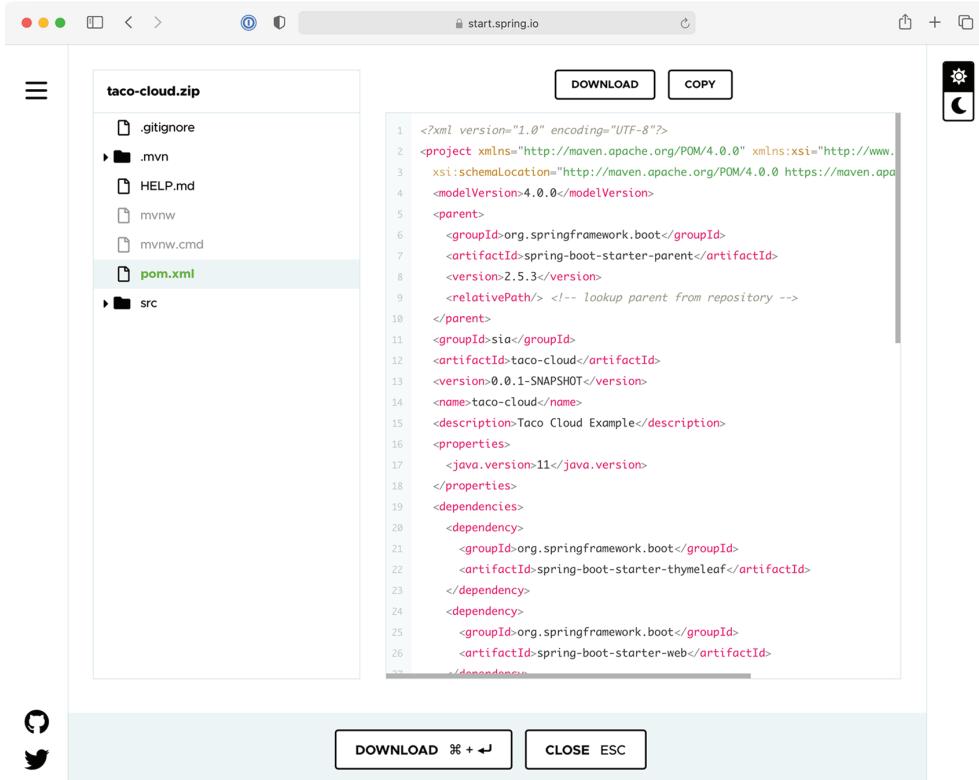


Рис. A.18 Полная версия пользовательского интерфейса Initializr

A.5 Инициализация проекта из командной строки

IDE и пользовательские веб-интерфейсы Spring Initializr являются, пожалуй, наиболее распространенными средствами создания и начальной инициализации проектов. Все они являются всего лишь клиентами службы REST, предлагаемой приложением Initializr. Но иногда в особых случаях (например, в сценариях) может пригодиться возможность использовать службу Initializr непосредственно из командной строки.

Вот два основных способа использования API службы:

- с помощью команды curl (или аналогичного клиента командной строки);
- с помощью интерфейса командной строки Spring Boot (Spring Boot CLI).

Рассмотрим оба способа и начнем с команды curl.

CURL И INITIALIZR API

Самый простой способ сгенерировать проект Spring с помощью curl – выполнить следующую команду:

```
% curl https://start.spring.io/starter.zip -o demo.zip
```

Эта команда обратится к конечной точке */starter.zip* службы Initializr, которая создаст проект Spring и загрузит его в виде zip-файла. Для сгенерированного проекта будет выбрана система сборки Maven, и в нем не будет никаких зависимостей, кроме базовой начальной зависимости Spring Boot. Во всех параметрах в файле *pom.xml* будут установлены значения по умолчанию.

Если не указать иное, то файл получит имя *starter.zip*. Но в этом примере указан параметр *-o*, задающий имя загружаемого файла *demo.zip*. Общедоступный сервер Spring Initializr находится по адресу <https://start.spring.io>, но если вы используете свой экземпляр Initializr, то измените URL в команде, подставив адрес своего экземпляра.

Возможно, вам понадобится задать еще ряд деталей и зависимостей, кроме заданных по умолчанию. В табл. А.1 перечислены все параметры (и их значения по умолчанию), которые можно задать при использовании службы Spring Initializr REST.

Таблица А.1 Параметры запроса, поддерживаемые Initializr API

Параметр	Описание	Значение по умолчанию
groupId	Идентификатор группы проекта для организации в репозитории Maven	com.example
artifactId	Идентификатор артефакта проекта, под которым он будет отображаться в репозитории Maven	demo
version	Версия проекта	0.0.1-SNAPSHOT
name	Название проекта; также используется для определения имени главного класса приложения (с суффиксом Application)	demo
description	Описание проекта	Demo project for Spring Boot
packageName	Имя базового пакета проекта	com.example.demo
dependencies	Зависимости для включения в спецификацию сборки проекта	Начальная зависимость Spring Boot
type	Тип генерируемого проекта: maven-project или gradle-project	maven-project
javaVersion	Используемая версия Java	1.8
bootVersion	Используемая версия Spring Boot	Текущая доступная версия Spring Boot
language	Используемый язык программирования: java, groovy или kotlin	java
packaging	Как проект должен упаковываться: jar или war	jar
applicationName	Имя приложения	Значение параметра name
baseDir	Имя базового каталога в сгенерированном архиве	Корневой каталог

Список этих параметров и доступных зависимостей можно также получить, отправив простой запрос по базовому URL:

```
% curl https://start.spring.io
```

Параметр `dependencies` – самый, пожалуй, востребованный. Например, предположим, что вы решили создать простой веб-проект. Следующая команда `curl` создаст ZIP-файл проекта с начальной зависимостью `web`:

```
% curl https://start.spring.io/starter.zip \
-d dependencies=web \
-o demo.zip
```

Более сложный пример: предположим, что вы решили написать веб-приложение, применяющее Spring Data JPA для хранения данных, использовать систему сборки Gradle и поместить проект в каталог с именем `my-dir`. Также предположим, что вместо загрузки zip-файла вы хотите, чтобы проект был сразу же распакован в файловую систему. В этом случае вам поможет следующая команда:

```
% curl https://start.spring.io/starter.tgz \
-d dependencies=web,data-jpa \
-d type=gradle-project \
-d baseDir=my-dir | tar -xvf -
```

Здесь загруженный zip-файл передается команде `tar` для распаковки.

Интерфейс командной строки Spring Boot

Интерфейс командной строки Spring Boot – еще один способ генерировать проект приложения Spring. Установить Spring Boot CLI можно разными способами, но самый простой способ (и мой любимый) – использовать SDKMAN (<http://sdkman.io/>), как показано ниже:

```
% sdk install springboot
```

После установки можно начинать использовать Spring Boot CLI для создания проектов. Основная команда, которую вы будете использовать: `spring init`. Вот как выглядит самая простая команда создания проекта с помощью Spring Boot CLI:

```
% spring init
```

Она загрузит zip-файл `demo.zip` с простым проектом Spring Boot. Однако часто желательно указать дополнительные параметры и зависимости. В табл. А.2 перечислены все параметры, доступные для команды `spring init`.

Список этих параметров и доступных зависимостей можно также получить, указав параметр `--list`:

```
% spring init --list
```

Таблица A.2 Параметры запроса, поддерживаемые командой `spring init`

Параметр	Описание	Значение по умолчанию
<code>group-id</code>	Идентификатор группы проекта для организации в репозитории Maven	<code>com.example</code>
<code>artifact-id</code>	Идентификатор артефакта проекта, под которым он будет отображаться в репозитории Maven	<code>demo</code>
<code>version</code>	Версия проекта	<code>0.0.1-SNAPSHOT</code>
<code>name</code>	Название проекта; также используется для определения имени главного класса приложения (с суффиксом Application)	<code>demo</code>
<code>description</code>	Описание проекта	<code>Demo project for Spring Boot</code>
<code>package-name</code>	Имя базового пакета проекта	<code>com.example.demo</code>
<code>dependencies</code>	Зависимости для включения в спецификацию сборки проекта	Начальная зависимость Spring Boot
<code>type</code>	Тип генерируемого проекта: <code>maven-project</code> или <code>gradle-project</code>	<code>maven-project</code>
<code>java-version</code>	Используемая версия Java	<code>1.8</code>
<code>boot-version</code>	Используемая версия Spring Boot	Текущая доступная версия Spring Boot
<code>language</code>	Используемый язык программирования: <code>java</code> , <code>groovy</code> или <code>kotlin</code>	<code>java</code>
<code>packaging</code>	Как проект должен упаковываться: <code>jar</code> или <code>war</code>	<code>jar</code>

Допустим, вы решили создать веб-приложение, основанное на Java 1.7. Для этого можно использовать следующую команду с параметрами `--dependencies` и `--java`:

```
% spring init --dependencies=web --java-version=1.7
```

Более сложный пример: предположим, что вы решили написать веб-приложение, применяющее Spring Data JPA для хранения данных, и использовать систему сборки Gradle вместо Maven. В этом случае вам поможет следующая команда:

```
% spring init --dependencies=web,jpa --type=gradle-project
```

Обратите также внимание, что многие параметры команды `spring init` очень похожи на параметры, используемые в команде `curl`. Однако стоит отметить, что `spring init` поддерживает не все параметры (например, она не поддерживает параметр `baseDir`), а кроме того, вместо «верблюжьей» нотации для разделения слов в именах параметров используется дефис (например, `package-name` вместо `packageName`).

A.6 Сборка и запуск проектов

Независимо от способа инициализации проекта, вы всегда можете запустить приложение из командной строки командой `java -jar`:

```
% java -jar demo.jar
```

Этот прием применим, даже если вы выбрали способ упаковки проекта в файл WAR вместо JAR:

```
% java -jar demo.war
```

Также можно воспользоваться плагинами Spring Boot для Maven и Gradle и запускать приложение с его помощью. Например, если сборка проекта производится с помощью Maven, то его можно запустить так:

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

Если, напротив, вы выбрали систему сборки Gradle, то команда запуска проекта будет выглядеть так:

```
% gradle bootRun
```

В любом случае, независимо от используемой системы сборки (Maven или Gradle), инструмент сборки сначала соберет проект (если он еще не собран) и запустит его.

Предметный указатель

Символы

- imageName, параметр, 498
- _typeId, свойство, 258
- <artifactId>, свойство, 497
- <dependencies>, элемент, 37
- <div>, элемент, 64
- <form>, элемент, 68, 166
- <input>, элемент, 64
- <p>, элемент, 64
- <packaging>, элемент, 507
- <parent>, элемент, 37
- , элемент, 64
- <version>, свойство, 497
- <version>, элемент, 37
- @Bean, аннотация, 28, 112, 174, 360
- @ComponentScan, аннотация, 39
- @ConditionalOnMissingBean, аннотация, 444
- @Configuration, аннотация, 28, 194, 360
- @ConfigurationProperties, аннотация, 182, 274, 309
- @CrossOrigin, аннотация, 200
- _csrf, атрибут, 166
- _csrf, поле, 166
- @Data, аннотация, 57
- @DataMongoTest, аннотация, 405
- @DataR2dbcTest, аннотация, 391, 398
- @DeleteOperation, аннотация, 466
- @DestinationVariable, аннотация, 421
- @DirtiesContext, аннотация, 399
- @Document, аннотация, 138, 400
- @EnableAdminServer, аннотация, 474
- @EnableAutoConfiguration, аннотация, 38
- @EnableWebSecurity, аннотация, 377
- @Endpoint, аннотация, 466
- @Field, аннотация, 138
- @Filter, аннотация, 294
- @GeneratedValue, аннотация, 117
- @GetMapping, аннотация, 68, 198, 200, 208
- @Header, аннотация, 285
- @Id, аннотация, 110, 116, 138
- @ImportResource, аннотация, 287
- @InboundChannelAdapter, аннотация, 305
- @JmsListener, аннотация, 262
- @JmxEndpoint, аннотация, 469
- @KafkaListener, аннотация, 281
- @ManagedAttribute, аннотация, 488
- @ManagedOperation, аннотация, 488
- @ManagedResource, аннотация, 488
- @ManyToMany, аннотация, 117
- @ManyToOne, аннотация, 170
- @MessageMapping, аннотация, 420
- @ModelAttribute, аннотация, 69
- @NoArgsConstructor, аннотация, 116
- @NotBlank, аннотация, 77
- @OneToMany, аннотация, 118
- @Order, аннотация, 231
- @PatchMapping, аннотация, 206
- @PathVariable, аннотация, 203, 421
- @PostAuthorize, аннотация, 169
- @PostMapping, аннотация, 68, 205
- @PreAuthorize, аннотация, 168
- @PrimaryKey, аннотация, 129
- @PrimaryKeyColumn, аннотация, 131
- @Profile, аннотация, 194
- @PutMapping, аннотация, 207
- @ReadOperation, аннотация, 466
- @RequestBody, аннотация, 205
- @RequestMapping, аннотация, 62, 200
- @RequestScope, аннотация, 245

@RestController, аннотация, 199, 209
 @ServiceActivator, аннотация, 294
 @SessionAttributes, аннотация, 63
 @Slf4j, аннотация, 71
 @SpringBootApplication, аннотация, 38
 @SpringBootConfiguration, аннотация, 38, 398
 @SpringBootTest, аннотация, 40
 @Table, аннотация, 116, 129
 @Transformer, аннотация, 295
 @Transient, аннотация, 393
 @Transient, транзакция, 138
 @UserDefinedType, аннотация, 132
 @Valid, аннотация, 79
 @WebEndpoint, аннотация, 469
 @WriteOperation, аннотация, 466

A

AbstractMessageRouter, компонент, 297
 access, атрибут, 116
 access(), метод, 158
 action, атрибут, 68
 Actuator, 435
 включение и отключение конечных точек, 437
 защита конечных точек, 469
 использование конечных точек, 438
 исследование автоконфигурации, 443
 исследование маршрутизации запросов, 448
 мониторинг потоков, 452
 наблюдение за действиями приложения, 451
 наблюдение за обработкой HTTP-запросов, 451
 получение информации о конфигурации, 442
 получение информации о приложении, 439
 получение информации о работоспособности, 440
 получение метрик времени выполнения, 454
 получение отчета о компонентах, 443
 проверка свойств окружения и конфигурации, 445
 управление уровнями журналирования, 449
 настройка, 457
 внедрение информации о сборке в конечную точку /info, 458

конечная точка /info, 457
 определение своих индикаторов работоспособности, 463
 регистрация пользовательских метрик, 464
 создание пользовательских конечных точек, 466
 настройка базового пути, 436
 Actuator MBean
 использование, 484
 отправка уведомлений, 489
 создание, 487
 addIngredient(), метод, 244
 addIngredientsToModel(), метод, 97
 addNote(), метод, 468
 addScript(), метод, 174
 addScripts(), метод, 174
 addTaco(), метод, 60, 393
 addViewControllers(), метод, 82, 161
 Admin, сервер, 476
 автентификация с помощью Actuator, 482
 защита, 481
 исследование возможностей, 476
 исследование свойств окружения, 479
 обзор состояния приложения, 477
 просмотр ключевых метрик, 478
 просмотр уровней журналирования, 480
 регистрация клиентов, 475
 создание, 473
 Advanced Message Queuing Protocol (AMQP), 248
 Alert, класс, 426
 all(), операция, 347
 allIngredients(), метод, 449
 AMQP, 264
 добавление поддержки RabbitMQ в Spring, 266
 настройка конвертера сообщений, 270
 настройка свойств сообщений, 270
 отправка сообщений с помощью RabbitTemplate, 267
 пассивный прием сообщений из RabbitMQ, 275
 получение сообщений из RabbitMQ, 271
 получение сообщений с помощью RabbitTemplate, 271
 AMQP (Advanced Message Queuing Protocol), 248
 anonymous(), метод, 158
 any(), операция, 347
 ApiProperties, объект, 315

ApplicationArguments, параметр, 113
applicationConfig, источник свойств, 446
ApplicationRunner, 112
ApplicationRunner, компонент, 113, 422
args, параметр, 113
array, тип, 387
assertOrder(), метод, 398
authenticated(), метод, 158
authenticationServerSecurityFilterChain(),
метод, 231

B

block(), метод, 458
buffer(), операция, 344
build(), метод, 379
build/libs, каталог, 495

C

cascade, атрибут, 119
Cassandra, 124
create keyspace, 126
клUSTER узлов, 125
командная оболочка CQL, 125
коэффициент репликации, 126
моделирование данных, 128
определение репозиториев, 135
политика балансировки нагрузки, 127
пространство ключей, 125
реактивное хранилище данных, 407
 классы предметной области, 408
 создание реактивных
 репозиториев, 412
 тестирование реактивных
 репозиториев, 413
репозитории, 124
Cassandra DataStax, драйвер, 127
ccCVV, свойство, 77
ccExpiration, свойство, 77
ccNumber, свойство, 77
cf, инструмент командной строки, 495
class, атрибут, 80
cloud, профиль, 193
collectList(), операция, 346
collectMap(), операция, 346
CommandLineRunner, 112
CommandLineRunner, компонент, 112, 194,
201, 231
configuration(), метод, 377
ConfigurationPropertiesAutoConfiguration,
компонент, 445

configure(), метод, 378, 469, 507
configuredLevel, свойство, 450
consumes, атрибут, 205
ContentTypeDelegatingMessageConverter,
класс, 270
contexts, элемент, 443
contribute(), метод, 458
convert(), метод, 69
convertAndSend(), метод, 252
Converter, интерфейс, 69
count(), метод, 458
createdDate, свойство, 215
create keyspace, 126
create keyspace, команда, 408
CrudRepository, интерфейс, 119, 389
CSRF (Cross-Site Request Forgery –
подделка межсайтовых запросов), 166
curl, клиент командной строки, 224, 438
curl, команда, 523

D

DataSource, 174
DataStax Cassandra, драйвер, 127
DDD (Domain-Driven Design –
предметно-ориентированное
проектирование), 99
defaultRequestChannel, атрибут, 285
delayElements(), операция, 333
delaySubscription(), операция, 333
DELETE, запрос, 208
delete(), метод, 219, 372
deleteAllOrders(), метод, 167
deleteById(), метод, 208
deleteNote(), метод, 468
deleteOrder(), метод, 208
deliveryName, свойство, 110
deliveryZip, свойство, 121
denyAll(), метод, 158
dependencies, свойство, 443
description, свойство, 189
DesignTacoController, класс, 62
Destination, компонент, 255
Destination, объект, 255
dev, профиль, 194
DI (Dependency Injection – внедрение
 зависимостей), 26
distinct(), операция, 340
Docker, 125
docker build, команда, 496
doTransform(), метод, 313

E

effectiveLevel, свойство, 450
EmailOrder, класс, 313
EmailOrder, объект, 313
EmailProperties, класс, 309
EmailToOrderTransformer, класс, 311
EndpointRequest.toAnyEndpoint(), метод, 470
Errors, объект, 80
exception, тег, 456
exchangeToFlux(), метод, 375
exchangeToMono(), метод, 375
excluding(), метод, 470
expectBodyList(), метод, 366

F

failOnNoGitDirectory, свойство, 460
fields, свойство, 80
filename, параметр, 285
Files, тип, 290
FileWriterGateway, интерфейс, 292
FileWritingMessageHandler, компонент, 289
filter(), метод, 294
filter(), операция, 339
filterChain(), метод, 157
findAll(), метод, 95, 201, 244, 354, 389, 392
findByDeliveryZip(), метод, 120
findById(), метод, 95, 204, 389
findBySlug(), метод, 390
findByUsername(), метод, 153, 380
findByUserOrderByPlacedAtDesc(), метод, 182
firstWithSignal(), 335
flatMap(), операция, 341, 362, 392
force, атрибут, 116
from(), метод, 305
fromArray(), метод, 329, 430
fromIterable(), метод, 329
fromStream(), метод, 330
fullyAuthenticated(), метод, 158

G

GeneratedKeyHolder, тип, 102
GenericHandler, интерфейс, 315
GenericSelector, интерфейс, 295
GenericTransformer, интерфейс, 295
GET, запросы, 42, 62, 155, 198, 362
get(), метод, 204

GET(), метод, 360
getAuthorities(), метод, 152
getContent(), метод, 354
getForEntity(), метод, 218
getForObject(), метод, 218, 369
getImapUrl(), метод, 311
getMessageConverter(), метод, 268
getTacoCount(), метод, 488
gratuity, свойство, 428
GratitudeIn, класс, 428
GratitudeOut, класс, 428

H

H2, консоль, 49
handle(), метод, 281, 315
handleGreeting(), метод, 421
hasAnyAuthority(), метод, 158
hasAnyRole(), метод, 158
hasAuthority(), метод, 158, 238
hasIpAddress(), метод, 158
hasRole(), метод, 158
HATEOAS, 211
HealthIndicator, интерфейс, 463
helloRouterFunction(), метод, 360
home(), метод, 42
HomeController, класс, 81
HttpSecurity, объект, 157, 378
HttpStatus, объект, 374

I

id, параметр, 203
id, поле, 91, 104
increment(), метод, 488
info.contact.email, свойство, 439
info.contact.phone, свойство, 439
InfoContributor, интерфейс, 457
Ingredient, класс, 56, 91, 384
Ingredient, объект, 218, 372, 384
Ingredient, сущность, 212
ingredientId, параметр, 218
IngredientRepository, интерфейс, 93, 212
IngredientRepository.findById(), метод, 97
ingredientsController, компонент, 443
IngredientUDT, класс, 132, 410
inputChannel, атрибут, 293
insert, запрос, 103
IntelliJ IDEA, 513
interval(), метод, 332
int-file, пространство имен, 287
is*, методы, 152

J

Jackson2JsonMessageConverter, класс, 270
Java, объявление потоков интеграции, 288
java -jar, команда, 525
Java Message Service (JMS), 248
настройка, 249
настройка конвертера сообщений, 256
получение сообщений
 объявление приемников
 сообщений, 262
 с помощью JmsTemplate, 260
постобработка сообщений, 258
преобразование сообщений перед
отправкой, 256
 JmsTemplate, 251
javax.persistence, пакет, 116
JdbcIngredientRepository, компонент, 94
JDBC (Java Database Connectivity –
соединение с базами данных), 88
 JdbcTemplate, 92
 определение репозиториев, 93
 подготовка объектов данных
 к хранению, 91
 чтение и запись данных, 89
JdbcTemplate, 92
JdbcTemplate, класс, 89
JMS (Java Message Service), 248
 JmsTemplate, 251
 настройка, 249
 настройка конвертера сообщений, 256
 получение сообщений
 объявление приемников
 сообщений, 262
 с помощью JmsTemplate, 260
 постобработка сообщений, 258
 преобразование сообщений перед
 отправкой, 256
JMX (Actuator MBean)
использование, 484
отправка уведомлений, 489
создание, 487
JNDI (Java Naming and Directory Interface –
служба имен и каталогов Java), 178
JPA (Java Persistence API), 88
json(), метод, 365
JSON Web Key (JWK), 233
JSON Web Token (JWT), 227
just(), метод, 327

K

Kafka, 276

настройка в Spring, 277
отправка сообщений с помощью
 KafkaTemplate, 278
получение сообщений из Kafka, 281
kubectl, инструмент командной
строки, 500
kubectl port-forward, команда, 501
Kubernetes, развертывание в, 499

L

LineItem, объект, 299
lineItemSplitter(), метод, 300
List<Taco>, свойство, 393
ListBodySpec, объект, 366
LivenessState.BROKEN, событие, 505
LivenessState.CORRECT, событие, 505
loadUserByUsername(), метод, 149, 379
log(), операция, 345
Logger, объект, 71
logging.file.name, свойство, 181
logging.file.path, свойство, 181
logging.level.tacos, свойство, 192
Lombok, библиотека, 57

M

main(), метод, 39
management.endpoint.health.show-details,
свойство, 440
management.endpoints.web.exposure.
include, свойство, 437
management.endpoint.web.base-path,
свойство, 436
management.health.probes.enabled,
свойство, 503
management.info.git.mode, свойство, 461
management.web.endpoints.web.exposure.
include, свойство, 468
map(), операция, 324, 343, 395, 396, 421
mapRowToIngredient(), метод, 96
MarshallingMessageConverter, класс, 270
matches(), метод, 148
mergeWith(), операция, 332
Message, объект, 313
MessageHandler, интерфейс, 302
MessageHandler, компонент, 301
MessagingMessageConverter, класс, 270
MeterRegistry, компонент, 464
method, тер, 456
MockMvc, объект, 44
Model, объект, 63

MongoDB, 124, 136
 реактивное хранилище документов, 399
 определение реактивных
 репозиториев, 403
 определение типов документов, 400
 тестирование реактивных
 репозиториев, 404
MongoTemplate, компонент, 445
Mono, объекты, 324
Mono, тип, 417
Mono.just(), метод, 395

N

name, свойство, 65
NetBeans, 514
NoSQL, базы данных, 123
 Cassandra, 124
not(), метод, 158
notes(), метод, 468
Notification, объект, 489
NotificationPublisherAware,
 интерфейс, 489

O

OAuth 2, 223
OAuth2, 163
oauth2ResourceServer(), метод, 239
onAfterCreate(), метод, 465, 488
onNext(), метод, 322
OpenID Connect (OIDC), 163
order(), метод, 69
OrderController, класс, 73
orderForm(), метод, 71
orderForUser(), метод, 182
OrderMessagingService, интерфейс, 253
OrderProps, класс, 184
OrderProps, компонент, 184
OrderRepository, интерфейс, 101
OrderRepository, служба управления
 агрегатами, 393
OrderSplitter, компонент, 299
org.springframework.boot, свойство, 480
org.springframework.data.annotation,
 пакет, 116
origins, атрибут, 200

P

Page, объект, 355
page, параметр, 215

PageRequest, объект, 183
pageSize, свойство, 183
PasswordEncoder, компонент, 148
PATCH, запрос, 206
patchOrder(), метод, 207
path, атрибут, 205
permitAll(), метод, 158
Persistable, интерфейс, 116
placedAt, свойство, 182
PlacedAt, свойство, 121
Player, объект, 341
pollRate, свойство, 311
POST, запросы, 68, 168, 201
post(), метод, 372
postForEntity(), метод, 220
postForLocation(), метод, 220
postForObject(), метод, 220
postProcessMessage(), метод, 271
postTaco(), метод, 205, 362
prefix, атрибут, 309
Processor, интерфейс, 322, 323
processOrder(), метод, 73, 106, 170
processPayload(), метод, 314
processRegistration(), метод, 156
processTaco(), метод, 68
prod, профиль, 192
produces, атрибут, 200
property, поле, 447
propertySources, поле, 446
provider, свойство, 242
publish(), метод, 505
Publisher, интерфейс, 322
PUT, запрос, 206
put(), метод, 219, 372
putOrder(), метод, 207

Q

qa, профиль, 194
query(), метод, 95
QueueChannel, компонент, 294

R

R2DBC (Reactive Relational Database Connectivity), 383
 реактивные репозитории, 389
 служба управления агрегатами
 в **OrderRepository**, 393
 сущности для R2DBC, 384
 тестирование репозиториев, 391
RabbitMQ, 264

- добавление поддержки RabbitMQ в Spring, 266
настройка конвертера сообщений, 270
настройка свойств сообщений, 270
отправка сообщений с помощью `RabbitTemplate`, 267
пассивный прием сообщений из RabbitMQ, 275
получение сообщений из RabbitMQ, 271
получение сообщений с помощью `RabbitTemplate`, 271
`range()`, метод, 330
`ReactiveCrudRepository`, интерфейс, 389
Reactive Streams, 321
`ReactiveUserDetailsService`, компонент, 380
Reactor
диаграммы реактивных потоков, 325
добавление зависимостей, 326
зависимости, 326
операции с реактивными потоками, 327
генерирование новых данных в потоке `Flux`, 330
комбинирование реактивных типов, 332
логические операции, 347
создание реактивных типов из коллекций, 329
реактивное программирование, 319
`ReadinessState.ACCEPTING_TRAFFIC`, событие, 505
`ReadinessState.REFUSING_TRAFFIC`, событие, 505
`receiveAndConvert()`, метод, 260
`recent()`, метод, 362
`recentTacos()`, метод, 201, 355
`RegisteredClientRepository`, интерфейс, 232
`registerForm()`, метод, 155
`RegistrationController`, класс, 154
`RegistrationForm`, объект, 156
`rememberMe()`, метод, 158
`replicas`, свойство, 500
`resource`, свойство, 443
`ResourceDatabasePopulator`, объекты, 388
`ResponseEntity`, объект, 200, 218
REST
безопасность
защита API с помощью сервера ресурсов, 238
разработка клиента, 241
создание сервера авторизации, 229
OAuth 2, 223
использование служб, 215
отправка ресурса запросом POST, 220
отправка ресурса запросом PUT, 219
получение ресурса запросом GET, 217
создание контроллеров RESTful, 198
извлечение данных с сервера, 198
изменение данных на сервере, 205
отправка данных на сервер, 204
удаление данных на сервере, 208
удаление ресурса запросом DELETE, 219
услуги на основе данных, 209
`RestTemplate`, класс, 217
`retrieve()`, метод, 369
`rootProject.name`, свойство, 495
`route()`, метод, 360
`routerFunction()`, метод, 362
`RSocket`
создание клиентов и серверов, 420
дву направленная передача сообщений, 427
модель запрос–ответ, 420
модель запрос–поток, 423
модель «запустил и забыл», 426
`RSocket`, обзор, 417
`RSocketRequester`, объект, 422
`run()`, метод, 39, 112
`RxJava`, типы, 357
-
- S**
- `save()`, метод, 96, 358, 392
`saveAll()`, метод, 358, 395
`saveIngredientRefs()`, метод, 104
`saveTaco()`, метод, 104
`Schedulers`, класс, 343
`SecurityConfig`, класс, 238
`SecurityFilterChain`, компонент, 157, 241
`SecurityWebFilterChain()`, метод, 379
`send()`, метод, 252
`sendNotification()`, метод, 489
`sendOrder()`, метод, 253
`SerializerMessageConverter`, класс, 270
`server.port`, свойство, 178, 447
`server.shutdown`, свойство, 501
`server.ssl.key-password`, свойство, 179
`server.ssl.key-store`, свойство, 179
`server.ssl.key-store-password`, свойство, 179
`setNotificationPublisher()`, метод, 489
`setup()`, метод, 399
`setViewName()`, метод, 83
`shouldReturnRecentTacos()`, метод, 364
`shouldSaveAndFetchIngredients()`, метод, 392

shouldSaveAndFetchOrders(), метод, 407
showDesignForm(), метод, 63
SimpleMessageConverter, класс, 270
size, параметр, 215
skip(), операция, 337
sort, параметр, 215
source, свойство, 258
sources(), метод, 507
Spring
 безопасность веб-запросов
 использование сторонних систем аутентификации, 163
 создание страницы входа, 160
 безопасность на уровне методов, 167
 безопасность приложений, 145
 включение Spring Security, 145
 знай своего пользователя, 169
 инициализация проекта из командной строки, 522
 контейнер, 26
 контекст приложения, 26
 настройка аутентификации, 147
 настройка аутентификации пользователя, 150
 предотвращение подделки межсайтовых запросов, 166
 развертывание приложений
 варианты, 493
 включение проверок готовности и жизнеспособности приложения, 503
 выполняемые файлы JAR, 494
 корректное завершение работы, 501
 образы контейнеров, 495
 проверка готовности и жизнеспособности приложения, 502
 развертывание в Kubernetes, 499
 создание и развертывание файлов WAR, 506
 разработка приложений, 41
 контроллеры
 тестирование, 43
 Spring Boot DevTools, 47
 сборка и запуск проектов, 525
 структуря проекта, 34
 хранение сведений о пользователях в памяти, 149
 IntelliJ IDEA, 513
 NetBeans, 514
 Spring Tool Suite, 509
 start.spring.io, 519
spring.activemq.broker-url, свойство, 250
spring.activemq.in-memory, свойство, 250
spring.activemq.password, свойство, 250
spring.activemq.user, свойство, 250
SpringApplication класс, 39
spring.application.name, свойство, 476
Spring Batch, 52
Spring Boot, поддерживающие механизмы шаблонов, 84
Spring Boot Actuator, 435
 включение и отключение конечных точек, 437
 защита конечных точек, 469
 использование конечных точек, 438
 исследование автоконфигурации, 443
 исследование маршрутизации запросов, 448
 мониторинг потоков, 452
 наблюдение за действиями приложения, 451
 наблюдение за обработкой HTTP-запросов, 451
 получение информации о конфигурации, 442
 получение информации о приложении, 439
 получение информации о работоспособности, 440
 получение метрик времени выполнения, 454
 получение отчета о компонентах, 443
 проверка свойств окружения и конфигурации, 445
 управление уровнями журналирования, 449
 конечная точка /info, 457
 внедрение информации о сборке, 458
 настройка, 457
 определение своих индикаторов работоспособности, 463
 регистрация пользовательских метрик, 464
 создание пользовательских конечных точек, 466
 настройка базового пути, 436
spring.boot.admin.client.password, свойство, 482
spring.boot.admin.client.url, свойство, 475
spring.boot.admin.client.username, свойство, 482
spring-boot-starter-security, начальная зависимость, 145
Spring Cloud, 53
Spring Data, 51

- Spring Data Cassandra, 124
 - включение в проект, 124
 - отображение типов данных для хранения в Cassandra, 129
 - spring-boot-starter-data-cassandra, 124
 - spring.data.cassandra.contact-points, свойство, 127
 - spring.data.cassandra.keyspace-name, свойство, 126
 - spring.data.cassandra.local-datacenter, свойство, 127
 - spring.data.cassandra.password, свойство, 127
 - spring.data.cassandra.port, свойство, 127
 - spring.data.cassandra.schema-action, свойство, 126
 - spring.data.cassandra.username, свойство, 127
 - spring.data.cassandra.contact-points, свойство, 127
 - spring.data.cassandra.keyspace-name, свойство, 126
 - spring.data.cassandra.local-datacenter, свойство, 127
 - spring.data.cassandra.password, свойство, 127
 - spring.data.cassandra.port, свойство, 127
 - spring.data.cassandra.schema-action, свойство, 126
 - spring.data.cassandra.username, свойство, 127

Spring Data JDBC, 106

 - аннотирование классов предметной области, 109
 - добавление в сборку, 107
 - интерфейсы репозиториев, 107
 - предварительная загрузка данных, 112

Spring Data JPA, 114

 - аннотирование классов данных, 115
 - добавление в сборку, 114
 - определение специализированных репозиториев, 120
 - хранение данных с, 114

Spring Data MongoDB

 - включение в проект, 137
 - отображение типов данных в документы, 138

spring-boot-starter-data-mongodb, 137

spring.data.mongodb.database, свойство, 138

spring.data.mongodb.host, свойство, 138

spring.data.mongodb.password, свойство, 138

spring.data.mongodb.port, свойство, 138

spring.data.mongodb.username, свойство, 138

spring.data.rest.base-path, свойство, 211

spring.datasource.data, свойство, 177

spring.datasource.driver-class-name, свойство, 177

spring.datasource.generate-unique-name, свойство, 93

spring.datasource.jndi-name cdjqcndj, 178

spring.datasource.name, свойство, 93

spring.datasource.schema, свойство, 177

Spring Expression Language (SpEL), 287

Spring Initializr веб-приложение, 29

Spring Integration, 52

 - ландшафт, 291
 - адаптеры каналов, 304
 - активаторы служб, 301
 - каналы сообщений, 292
 - маршрутизаторы, 296
 - модули конечных точек, 306
 - преобразователи, 295
 - сплиттеры, 298
 - фильтры, 294
 - шлюзы, 303

объявление потоков интеграции, 284

 - на Java, 288
 - на Java DSL, 290
 - на XML, 286

создание потока интеграции для электронной почты, 308

spring.jms.template.default-destination, свойство, 254

spring.kafka.bootstrap-servers, свойство, 278

spring.kafka.template.default-topic, свойство, 280

spring.main.web-application-type, свойство, 316

Spring MVC, 354

Spring Native, 53

spring.profiles.active, свойство, 192

spring.rabbitmq.addresses, свойство, 266

spring.rabbitmq.host, свойство, 266

spring.rabbitmq.password, свойство, 266

spring.rabbitmq.port, свойство, 266

spring.rabbitmq.template.exchange, свойство, 269

spring.rabbitmq.template.receive-timeout, свойство, 274

spring.rabbitmq.template.routing-key, свойство, 269

spring.rabbitmq.username, свойство, 266
 spring.rsocket.server.mapping-path, свойство, 431
 spring.rsocket.server.port, свойство, 421
 spring.rsocket.server.transport, свойство, 431
 Spring Security, 52, 145
 spring-boot-starter-security, начальная зависимость, 145
 Spring Tool Suite, 509
 Spring WebFlux, 351
 обзор, 352
 создание реактивных контроллеров, 354
 возврат одиночных значений, 356
 реактивная обработка ввода, 357
 типы RxJava, 357
 SQL-запросы, 89
 SQLException, исключение, 90
 start.spring.io, 519
 StockQuote, класс, 423
 strategy, атрибут, 117
 subscribe(), метод, 322
 subscribeOn(), операция, 342
 Subscriber, интерфейс, 322
 Subscription, интерфейс, 322
 systemEnvironment, источник свойств, 446

T

Taco, класс, 59, 91, 214, 386
 Taco, класс сущности, 385
 Taco, объект, 205, 357, 386
 Taco, параметр, 205
 Taco, сущность, 212
 TacoCloudApplication, класс, 38, 507
 TacoCloudApplicationTests, класс, 44
 tacocloud.ingredients, пакет, 450
 tacoIds, свойство, 393
 TacoOrder, класс, 59, 170, 258, 387
 TacoOrder, объект, 169, 313
 TacoOrder, сущность, 212
 TacoOrder, тип, 258
 TacoOrderAggregateService, класс, 396
 tacoOrderEmailFlow(), метод, 311
 taco.orders.pageSize, свойство, 184
 TacoRepository, интерфейс, 212
 TacoRepository.findAllById(), метод, 396
 tacos, пакет, 192
 tacos, свойство, 394
 TacoUDT, класс, 135, 411
 tag, атрибут запроса, 456

take(), операция, 338, 355
 target, каталог, 494
 testHomePage(), метод, 44
 testTaco(), метод, 364
 text, свойство, 468
 th:action, атрибут, 166
 th:each, атрибут, 64
 th:errors, атрибут, 80
 th:field, атрибут, 65
 th:if, атрибут, 80
 th:src, атрибут, 43
 th:text, атрибут, 65
 th:value, атрибут, 64
 Thymeleaf механизм шаблонов, 42
 toRoman(), метод, 295
 toUser(), метод, 156
 transform(), метод, 296
 Transformer, интерфейс, 296
 transformFlow(), метод, 296
 type, атрибут, 131

U

update(), метод, 96
 uppercase(), метод, 304
 UpperCaseGateway, интерфейс, 304
 URI базовый, 370
 User, класс, 151
 User, тип, 152
 UserDetails, интерфейс, 152
 UserDetails, объект, 149, 380
 UserDetailsService, интерфейс, 149
 UserDetailsService, компонент, 149, 230
 userDetailsService(), метод, 153
 user-info-uri, свойство, 243
 UUID, тип данных, 131

V

ViewControllerRegistry, класс, 83

W

war, плагин, 507
 WebClient, компонент, 370
 WebClient, объект, 369
 WebConfig, класс, 82
 webEnvironment, атрибут, 368
 WebMvcConfigurer, интерфейс, 82
 WebSecurityConfigurerAdapter, интерфейс, 377
 WebSocket, 431

websocket(), метод, 432
WebTestClient, объект, 366
withDetail(), метод, 458
writeToFile(), метод, 285

Z

zip(), операция, 333

A

Автоконфигурация, 28, 174
абстракция окружения Spring, 175
встроенный сервер, 178
журналирование, 179
использование специальных значений
свойств, 181
настройка источника данных, 177
обзор, 443
Автоматический перезапуск
приложения, 47
Автоматическое обновление браузера, 48
Автоматическое отключение
кеширования шаблонов, 48
Автоматическое связывание, 28
Адаптеры каналов, 304
Администрирование Spring
 Spring Boot Admin, 473
 аутентификация с помощью
 Actuator, 482
 возможности, 476
 защита, 481
 исследование свойств окружения, 479
 обзор состояния приложения, 477
 просмотр ключевых метрик, 478
 просмотр уровней
 журналирования, 480
 регистрация клиентов, 475
 создание сервера Admin, 473
Активаторы служб, 301
Активация профилей, 192
Аннотирование классов предметной
области, 109
Асинхронный обмен сообщениями
 через JMS, 248
 настройка, 249
 объявление приемников
 сообщений, 262
 получение сообщений, 260
 JmsTemplate, 251
 через Kafka, 276
 настройка в Spring, 277

отправка сообщений с помощью
KafkaTemplate, 278
получение сообщений из Kafka, 281
через RabbitMQ и AMQP, 264
 добавление поддержки RabbitMQ
 в Spring, 266
 настройка конвертера
 сообщений, 270
 настройка свойств сообщений, 270
 отправка сообщений с помощью
 RabbitTemplate, 267
 пассивный прием сообщений из
 RabbitMQ, 275
 получение сообщений из
 RabbitMQ, 271
 получение сообщений с помощью
 RabbitTemplate, 271
Аутентификация, 147
 настройка аутентификации
 пользователя, 150
 хранение сведений о пользователях
 в памяти, 149

B

Базовый путь, Actuator, 436
Базовый URI, 370
Безопасность
 REST
 защита API с помощью сервера
 ресурсов, 238
 разработка клиента, 241
 создание сервера авторизации, 229
 OAuth 2, 223
 Spring
 включение Spring Security, 145
 знай своего пользователя, 169
 настройка аутентификации, 147
 настройка аутентификации
 пользователя, 150
 веб-запросов, 157
 обзор, 157
 веб-запросы
 использование сторонних систем
 аутентификации, 163
 страницы входа, 160
 на уровне методов, 167
 предотвращение подделки
 межсайтовых запросов, 166
 реактивных API, 377
Буферизация данных в реактивном
потоке, 344

В

Веб-запросы, 41
 защита, 157
 использование сторонних систем аутентификации, 163
 страницы входа, 160

Веб-приложения
 отображение информации, 55
 выполнение проверки после привязки, 79
 обработка форм, 68
 объявление правил для проверки данных, 76
 определение предметной области, 56
 проверка данных в формах, 75
 создание класса контроллера, 60
 создание представления, 64

Включение и отключение конечных точек, 437

Включение проверок жизнеспособности и готовности, 503

Внедрение зависимостей (Dependency Injection, DI), 26

Выбор механизма шаблонов для создания представлений, 84

Выполнение длительных запросов, 371

Г

Готовность и жизнеспособность приложения, 503

Д

Данные
 буферизация в реактивном потоке, 344
 контроллеры RESTful
 извлечение данных с сервера, 198
 изменение данных на сервере, 205
 отправка данных на сервер, 204
 удаление данных на сервере, 208
 моделирование в Cassandra, 128
 отображение реактивных данных, 341
 реактивное хранилище документов, 399
 фильтрация в реактивных потоках, 336

Двунаправленная передача сообщений, серверы, 427

Действующие серверы, 367

Диаграммы реактивных потоков, 325

Длительные запросы, выполнение, 371

Документные данные, 399

Ж

Журналирование, управление уровнями, 449

З

Загрузка приложения, 38
 Запрос–ответ, серверы, 420
 Запрос–поток, серверы, 423
 «Запустил и забыл», серверы, 426
 Защита веб-запросов, 157
 Знай своего пользователя, 169

И

Инициализация приложений, 29
 загрузка, 38
 параметры сборки, 35
 структура проекта, 34
 тестирование приложения, 39

Инициализация проекта из командной строки, 522

Использование сторонних систем аутентификации, 163

К

Каналы сообщений, 292

Кеширование шаблонов, 85

Клиентское приложение, 228

Клиенты
 RSocket, создание, 419
 двунаправленная передача сообщений, 427
 модель запрос–ответ, 420
 модель запрос–поток, 423
 модель «запустил и забыл», 426
 разработка, 241

Коллекции, операции создания реактивных потоков, 329

Командная строка
 интерфейс командной строки Spring Boot, 524
 curl и Initializr API, 523

Комбинирование реактивных типов, 332
 объединение, 332

Компоненты, условное создание с помощью профилей, 193

Конвертер сообщений JmsTemplate
 настройка, 256
 перед отправкой, 256

- Конечные точки, 438
/info, 457
внедрение информации о сборке, 458
включение и отключение, 437
защита, 469
модули, 306
получение информации о приложении, 439
исследование автоконфигурации, 443
исследование маршрутизации запросов, 448
конфигурация, 442
мониторинг потоков, 452
наблюдение за действиями приложения, 451
наблюдение за обработкой HTTP-запросов, 451
получение метрик времени выполнения, 454
получение отчета о компонентах, 443
проверка работоспособности, 440
проверка свойств окружения и конфигурации, 445
управление уровнями журнализирования, 449
создание пользовательских конечных точек, 466
Контейнер Spring, 26
Контекст приложения Spring, 26
Контроллеры
создание класса, 60
создание контроллеров RESTful, 198
извлечение данных с сервера, 198
изменение данных на сервере, 205
отправка данных на сервер, 204
удаление данных на сервере, 208
создание реактивных, 354
возврат одиночных значений, 356
использование типов RxJava, 357
реактивная обработка ввода, 357
тестирование, 43
Контроллеры представлений, 82
Конфигурационные свойства
автоконфигурация, 174
абстракция окружения Spring, 175
встроенный сервер, 178
журнализование, 179
использование специальных значений свойств, 181
настройка источника данных, 177
и профили, 190
определение свойств профилей, 191
создание, 182
объявление метаданных, 187
определение хранителей конфигурационных свойств, 184
Корректное завершение работы, 501
-
- Л**
- Логические операции, 347
-
- М**
- Маршрутизаторы, 296
Метаданные, конфигурационных свойств, 187
Метрики
времени выполнения, получение, 454
регистрация, 464
Микросервисы, 52
Моделирование данных в Cassandra, 128
Модель активного извлечения (pull), 260
Модель пассивного получения (push), 260
Модули конечных точек, 306
Мониторинг потоков выполнения, 452
Мониторинг Spring Actuator MBean
использование, 484
отправка уведомлений, 489
создание, 487
-
- Н**
- Наблюдение за действиями приложения, 451
Настройка имен отношений и путей к ресурсам, 212
Настройка источника данных, 177
Нереляционные базы данных, 123
Неявное предоставление разрешений, 227
-
- О**
- Обмен сообщениями
через JMS, 248
настройка, 249
объявление приемников сообщений, 262
получение сообщений, 260
JmsTemplate, 251
через Kafka, 276
настройка в Spring, 277

- отправка сообщений с помощью KafkaTemplate, 278
 - получение сообщений из Kafka, 281
 - через RabbitMQ и AMQP, 264
 - добавление поддержки RabbitMQ в Spring, 266
 - настройка конвертера сообщений, 270
 - настройка свойств сообщений, 270
 - отправка сообщений с помощью RabbitTemplate, 267
 - пассивный прием сообщений из RabbitMQ, 275
 - получение сообщений из RabbitMQ, 271
 - получение сообщений с помощью RabbitTemplate, 271
 - Обработка форм, 68
 - Образы контейнеров, 495
 - проверка готовности и жизнеспособности приложения, 502
 - настройка в процессе развертывания, 504
 - Объединение реактивных типов, 332
 - Объекты, операции создания реактивных потоков, 327
 - Операции комбинирования
 - выбор реактивного потока, первым отправившего событие, 335
 - Операции создания, 327
 - генерирование новых данных в потоке Flux, 330
 - из коллекций, 329
 - из объектов, 327
 - Операции с реактивными потоками, 327
 - комбинирование реактивных типов, 332
 - логические операции, 347
 - создание реактивных типов, 327
 - генерирование новых данных в потоке Flux, 330
 - из коллекций, 329
 - из объектов, 327
 - Определение репозиториев Cassandra, 135
 - Определение репозиториев JDBC, 93
 - Определение схемы и предварительная загрузка данных, 98
 - Отображение
 - веб-приложений, 55
 - определение предметной области, 56
 - выполнение проверки после привязки, 79
 - обработка форм, 68
 - проверка данных в формах, 75
 - объявление правил, 76
 - создание класса контроллера, 60
 - создание представления, 64
 - реактивных данных, 341
 - Отправка ресурса
 - запросом POST, 220
 - запросом PUT, 219
-
- П**
- Параметры сборки, 35
 - Подделка межсайтовых запросов (Cross-Site Request Forgery, CSRF), 166
 - Получение метрик времени выполнения, 454
 - Получение ресурса запросом GET, 217
 - Получение сообщений через JMS, 260
 - объявление приемников сообщений, 262
 - с помощью JmsTemplate, 260
 - Пользователи
 - знай своего пользователя, 169
 - настройка аутентификации пользователя, 150
 - определение типа данных и репозитория, 151
 - регистрация, 154
 - создание службы хранения учетных записей, 153
 - хранение сведений о пользователях в памяти, 149
 - Постобработка сообщений, 258
 - Потоки интеграции, 284
 - на Java, 288
 - на Java DSL, 290
 - на XML, 286
 - Поток интеграции для электронной почты, 308
 - Предметная область, определение, 56
 - Предметно-ориентированное проектирование (Domain-Driven Design, DDD), 99
 - Предметно-ориентированный язык (Domain-Specific Language, DSL), 120
 - Предметные области, сущности для R2DBC, 384

- Предоставление учетных данных клиента, 227
Предоставление учетных данных пользователя, 227
Предотвращение подделки межсайтовых запросов, 166
Преобразователи, 295
Приложения
 Spring
 IntelliJ IDEA, 513
 NetBeans, 514
 Spring Tool Suite, 509
 start.spring.io, 519
 инициализация проекта из командной строки, 522
 информация о
 исследование автоконфигурации, 443
 исследование маршрутизации запросов, 448
 конфигурация, 442
 мониторинг потоков, 452
 наблюдение за действиями приложения, 451
 наблюдение за обработкой HTTP-запросов, 451
 получение, 439
 получение метрик времени выполнения, 454
 получение отчета о компонентах, 443
 проверка работоспособности, 440
 проверка свойств окружения и конфигурации, 445
 управление уровнями журналирования, 449
Проверка данных в формах, 75
Профили, 190
 активация профилей, 192
 определение свойств профилей, 191
- P**
- Работа с данными, 88
 чтение и запись данных, 89
Развертывание приложений Spring
 варианты, 493
 включение проверок готовности и жизнеспособности приложения, 503
 выполняемые файлы JAR, 494
 корректное завершение работы, 501
 образы контейнеров, 495
 проверка готовности и жизнеспособности приложения, 502
 развертывание в Kubernetes, 499
 создание и развертывание файлов WAR, 506
Разработка приложений, 41
 контроллеры, тестирование, 43
 обработка веб-запросов, 41
 сборка и запуск, 45
 Spring Boot DevTools, 47
Реактивная служба учетных записей, 379
Реактивное программирование, 319
Реактивное хранение данных
 R2DBC, 383
 реактивные репозитории, 389
 служба управления агрегатами в OrderRepository, 393
 сущности для R2DBC, 384
 тестирование репозиториев, 391
 в Cassandra, 407
 классы предметной области, 408
 создание реактивных репозиториев, 412
 тестирование реактивных репозиториев, 413
 MongoDB
 определение реактивных репозиториев, 403
 определение типов документов, 400
 реактивное хранилище документов, 399
 тестирование реактивных репозиториев, 404
Реактивные API
 реактивные REST API, 369
 выполнение длительных запросов, 371
 защита, 376
 настройка реактивной службы учетных записей, 379
 обмен запросами, 375
 обработка ошибок, 373
 отправка ресурсов, 372
 получение ресурсов, 369
 удаление ресурсов, 372
 тестирование реактивных контроллеров, 363
 запросы GET, 363
 запросы POST, 366
 с действующим сервером, 367
 Spring WebFlux, 351

- обзор, 352
 реактивная обработка ввода, 357
 создание реактивных контроллеров, 354
 типы RxJava, 357
 функциональные обработчики запросов, 359
- Р**еактивные REST API, 369
 выполнение длительных запросов, 371
 защита, 376
 настройка реактивной службы учетных записей, 379
 обмен запросами, 375
 обработка ошибок, 373
 отправка ресурсов, 372
 получение ресурсов, 369
 удаление ресурсов, 372
 Регистрация пользователей, 154
 Реляционные базы данных, 89
 Репозитории
 Cassandra, 124
 реактивные, Cassandra, 412
-
- С**борка и запуск приложений, 45
 Сервер авторизации, 227
 Сервер ресурсов, 227, 238
 Серверы
 защита API с помощью сервера ресурсов, 238
 создание сервера авторизации, 229
 RSocket, создание
 двунаправленная передача сообщений, 427
 модель запрос–ответ, 420
 модель запрос–поток, 423
 модель «запустил и забыл», 426
 Сканирование компонентов, 28
 Служба имен и каталогов Java (Java Naming and Directory Interface, JNDI), 178
 Служба управления агрегатами в OrderRepository, 393
 Службы REST, 197
 Создание службы хранения учетных записей, 153
 Сохранение данных, 101
 Сплиттеры, 298
 Страницы вход, 160
 Структура проекта
- загрузка, 38
 параметры сборки, 35
 тестирование приложения, 39
 Структура проекта Spring, 34
-
- Т**естирование приложения, 39
 Тестирование реактивных контроллеров, 363
 запросы GET, 363
 запросы POST, 366
 с действующим сервером, 367
-
- У**даление ресурса запросом DELETE, 219
 Удаление ресурсов, 372
 Услуги на основе данных, 209
-
- Ф**ильтрация данных, в реактивных потоках, 336
 Фильтры, 294
 Формы
 обработка, 68
 проверка ввода, 75
 проверка данных
 объявление правил, 76
 Фреймворк без фреймворка, 49
 Функциональные обработчики запросов, 359
-
- Х**ранение данных, 88
 реактивное
 в Cassandra, 407
 R2DBC (Reactive Relational Database Connectivity), 383
 Хранение сведений о пользователях в памяти, 149
 Хранители конфигурационных свойств, 184
-
- Ц**икл обработки событий, 351

Ч

Чтение и запись данных, 89
Что такое Spring, 26

Ш

Шлюзы, 303

Э

Эрик Эванс (Eric Evans), 56

Я

Ядро Spring Framework, 50

Книги издательства «ДМК ПРЕСС»
можно купить оптом и в розницу
в книготорговой компании «Галактика»
(представляет интересы издательств
«ДМК ПРЕСС», «СОЛООН ПРЕСС», «КТК Галактика»).

Адрес: г. Москва, пр. Андропова, 38;
тел.: (499) 782-38-89, электронная почта: books@aliens-kniga.ru.

При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.

Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: <http://www.galaktika-dmk.com/>.

Крейг Уоллс

Spring в действии

6-е издание

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Зам. главного редактора *Сенченкова Е. А.*
Перевод *Киселев А. Н.*
Корректор *Синяева Г. И.*
Верстка *Чаннова А. А.*
Дизайн обложки *Мовчан А. Г.*

Гарнитура РТ Serif. Печать цифровая.
Усл. печ. л. 44,2. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com

Spring в действии

Каждый разработчик на Java должен быть знаком со Spring! Почему? Этот мощный фреймворк избавляет от множества утомительной работы, связанной с настройкой и решением повторяющихся задач, и упрощает создание готового к работе программного обеспечения промышленного качества. Последние обновления в значительной мере способствуют увеличению продуктивности программиста при разработке микросервисов, реактивных служб и других современных приложений. Неудивительно, что больше половины всех разработчиков на Java используют Spring.

Перед вами перевод **6-го издания** великолепной книги Крейга Уоллса. Шаг за шагом вы пройдете путь создания законченного веб-приложения на основе базы данных. Новое издание охватывает не только основы Spring, но и новые возможности, такие как реактивные потоки или интеграция с Kubernetes и RSocket.

Независимо от того, впервые ли вы знакомитесь с фреймворком Spring или переходите на новую версию Spring 5.3, этот классический бестселлер станет вашей настольной книгой!

Рассматриваемые темы:

- реляционные базы данных и базы данных NoSQL;
- интеграция со службами REST через RSocket;
- приемы реактивного программирования;
- развертывание приложений на традиционных серверах и в контейнерах.

Крейг Уоллс – старший инженер-разработчик в VMware, член команды разработчиков Spring, известный автор, частый участник конференций и активный популяризатор Spring.

Интернет-магазин: www.dmkpress.com

Оптовая продажа: КТК «Галактика»
books@aliens-kniga.ru



«Этой книги более чем достаточно для изучения и освоения экосистемы Spring. Настоятельно рекомендую прочитать!»
Пьер-Мишель Анセル, 8x8

«Лучший источник информации для современного разработчика, использующего Spring».
Бекки Хьюотт, Big Shovel Labs

«Искрывающее руководство для программистов, желающих создавать надежные, отказоустойчивые и масштабируемые облачные приложения с использованием Spring».
Дэвид Уизерспун, Parsons

«Spring продолжает развиваться! Приобретите это издание, чтобы продолжать развиваться вместе с фреймворком».
Кевин Ляо, Sotheby's

«Эта книга – ваш быстрый путь к освоению Spring Boot».
Дэвид Торрубия Иньиго,
MASMOVIL Group

ISBN 978-5-93700-112-2



9 785937 001122 >