

SPRING

БЫСТРО

ЛАУРЕНЦИУ СПИЛКЭ



MANNING

Spring Start Here

LEARN WHAT YOU NEED AND LEARN IT WELL

LAURENȚIU SPILCĂ
FOREWORD BY VICTOR RENTEA

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



MANNING
SHELTER ISLAND

ЛАУРЕНЦИУ СПИЛКЭ

SPRING

Быстро



Санкт-Петербург · Москва · Минск

2023

ББК 32.988.02-018
УДК 004.738.2
С72

Спилкэ Лауренциу

С72 Spring быстро. — СПб.: Питер, 2023. — 448 с.: ил.
ISBN 978-5-4461-1969-1

Java-программистам необходим фреймворк Spring. Этот невероятный инструмент универсален: вы можете разрабатывать как приложения для малого бизнеса, так и микросервисные архитектуры промышленного масштаба. Освоить Spring не так-то просто, но первый шаг сделать легко! Книга предназначена для Java-разработчиков, желающих создавать приложения на основе Spring. Информативные иллюстрации, понятные примеры, а также ясное и живое изложение Лауренциу Спилкэ позволят быстро овладеть необходимыми навыками. Вы научитесь планировать, разрабатывать и тестировать приложения. Благодаря акценту на наиболее важных функциях разберетесь в богатой экосистеме фреймворка Spring.

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

ББК 32.988.02-018
УДК 004.738.2

Права на издание получены по соглашению с Manning Publications. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

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

ISBN 978-1617298691 англ.
ISBN 978-5-4461-1969-1

© 2021 by Manning Publications Co. All rights reserved
© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022

Краткое содержание

Предисловие	14
Введение	16
Благодарности	18
О книге	20
Об авторе	24
Иллюстрация на обложке	25
От издательства	26

Часть I **Основные принципы**

Глава 1. Spring в реальном мире	28
Глава 2. Контекст Spring: что такое бины	50
Глава 3. Контекст Spring: создаем новые бины	80
Глава 4. Контекст Spring: использование абстракций.	106
Глава 5. Контекст Spring: области видимости и жизненный цикл бинов.	132
Глава 6. Аспекты и АОП в Spring	156

Часть II
Реализация

Глава 7. Введение в Spring Boot и Spring MVC.	190
Глава 8. Реализация веб-приложений с использованием Spring Boot и Spring MVC	216
Глава 9. Области веб-видимости бинов Spring	241
Глава 10. Реализация REST-сервисов	270
Глава 11. Использование конечных точек REST.	293
Глава 12. Использование источников данных в Spring-приложениях	315
Глава 13. Транзакции в Spring-приложениях.	337
Глава 14. Сохранение данных с помощью Spring Data	357
Глава 15. Тестирование Spring-приложений	379
Приложение А. Архитектурные концепции	404
Приложение Б. Использование XML в конфигурации контекста.	418
Приложение В. Краткое введение в HTTP	420
Приложение Г. Представление данных в формате JSON	431
Приложение Д. Установка MySQL и создание базы данных.	434
Приложение Е. Рекомендованные инструменты	442
Приложение Ж. Материалы, рекомендуемые для дальнейшего изучения Spring	444

Оглавление

Предисловие	14
Введение	16
Благодарности	18
О книге	20
Кто должен прочитать эту книгу	21
Структура издания	21
О коде	23
Об авторе	24
Иллюстрация на обложке	25
От издательства	26

Часть I **Основные принципы**

Глава 1. Spring в реальном мире	28
1.1. Зачем нужны фреймворки	29
1.2. Экосистема Spring	33
1.2.1. Spring Core вблизи: ядро Spring	34
1.2.2. Сохранение данных приложения с помощью Spring Data Access	36

8 Оглавление

1.2.3. Возможности Spring MVC для разработки веб-приложений	36
1.2.4. Тестирование в Spring	36
1.2.5. Проекты на базе экосистемы Spring	37
1.3. Spring в реальных задачах	39
1.3.1. Использование Spring для разработки серверных приложений.	39
1.3.2. Использование Spring для средств автоматизации тестирования	41
1.3.3. Использование Spring для разработки десктопных приложений.	43
1.3.4. Использование Spring в мобильных приложениях	44
1.4. Когда не стоит использовать фреймворки	44
1.4.1. Приложение должно занимать как можно меньше места	45
1.4.2. Безопасность требует написать весь код самостоятельно	45
1.4.3. Использовать фреймворк нецелесообразно из-за слишком большого количества настроек	46
1.4.4. Когда переход на фреймворк не приносит пользы	46
1.5. Чему вы научитесь, прочитав эту книгу.	48
Резюме	48
Глава 2. Контекст Spring: что такое бины	50
2.1. Создание проекта Maven	51
2.2. Добавление бинов в контекст Spring.	57
2.2.1. Добавление бинов в контекст Spring с помощью аннотации @Bean	61
2.2.2. Добавление бинов в контекст Spring с помощью стереотипных аннотаций	70
2.2.3. Программное добавление бинов в контекст Spring	75
Резюме	79
Глава 3. Контекст Spring: создаем новые бины	80
3.1. Установка связей между бинами, описанными в файле конфигурации	82
3.1.1. Монтаж бинов путем прямого вызова одного метода с аннотацией @Bean из другого такого же метода	85

3.1.2. Монтаж бинов путем передачи параметра в метод с аннотацией @Bean.	89
3.2. Внедрение бинов с помощью аннотации @Autowired	91
3.2.1. Внедрение значений через поля класса с использованием аннотации @Autowired	92
3.2.2. Использование аннотации @Autowired для внедрения значения через конструктор.	95
3.2.3. Внедрение зависимости через сеттер	96
3.3. Циклические зависимости	97
3.4. Выбор из нескольких бинов в контексте Spring.	99
Резюме	104
Глава 4. Контекст Spring: использование абстракций.	106
4.1. Применение интерфейсов для определения контрактов	107
4.1.1. Использование интерфейсов для разделения реализаций	107
4.1.2. Условия задачи	111
4.1.3. Реализация сценариев использования без применения фреймворка	111
4.2. Использование внедрения зависимостей для абстракций	117
4.2.1. Выбор объектов для добавления в контекст Spring.	117
4.2.2. Выбор одной из реализаций абстракции для автомонтажа	123
4.3. Подробнее об обязанностях объектов со стереотипными аннотациями	129
Резюме	130
Глава 5. Контекст Spring: области видимости и жизненный цикл бинов.	132
5.1. Использование одиночной области видимости	133
5.1.1. Что такое одиночный бин.	133
5.1.2. Одиночные бины в реальных приложениях	141
5.1.3. Немедленное и «ленивое» создание экземпляров.	143
5.2. Прототипная область видимости бинов.	145
5.2.1. Как работают прототипные бины.	146
5.2.2. Практическое применение прототипных бинов.	150
Резюме	155

10 Оглавление

Глава 6. Аспекты и АОП в Spring	156
6.1. Аспекты в Spring	158
6.2. Реализация аспектов в Spring с помощью АОП.	162
6.2.1. Реализация простого аспекта	162
6.2.2. Изменение параметров и возвращаемого значения перехваченного метода	171
6.2.3. Перехват методов с аннотациями.	176
6.2.4. Другие полезные аннотации советов.	180
6.3. Щепочки выполнения аспектов	181
Резюме	188

Часть II **Реализация**

Глава 7. Введение в Spring Boot и Spring MVC.	190
7.1. Что такое веб-приложение	191
7.1.1. Основные сведения о веб-приложениях	192
7.1.2. Способы реализации веб-приложений на основе Spring	193
7.1.3. Использование контейнера сервлетов в веб-разработке.	196
7.2. Магия Spring Boot	199
7.2.1. Создание проекта Spring Boot с помощью сервиса инициализации проекта	200
7.2.2. Упрощенное управление зависимостями с помощью диспетчеров зависимостей	206
7.2.3. Автоматическая конфигурация по соглашению на основе зависимостей	208
7.3. Реализация приложения с помощью Spring MVC	208
Резюме	214

Глава 8. Реализация веб-приложений с использованием Spring Boot и Spring MVC.	216
--	-----

8.1. Создание веб-приложений с динамическими представлениями	217
8.1.1. Получение данных из HTTP-запроса	222
8.1.2. Передача данных от клиента серверу посредством параметров запроса	223

8.1.3. Передача данных от клиента серверу с помощью переменных пути	227
8.2. Использование HTTP-методов GET и POST	229
Резюме	239
Глава 9. Области веб-видимости бинов Spring	241
9.1. Использование бинов с областью видимости в рамках запроса в веб-приложениях Spring.	242
9.2. Использование области видимости в рамках сессии в веб-приложениях Spring.	253
9.3. Использование области видимости в рамках всего веб-приложения Spring.	264
Резюме	268
Глава 10. Реализация REST-сервисов	270
10.1. Обмен данными между приложениями посредством REST-сервисов.	272
10.2. Создание конечной точки REST.	274
10.3. Управление HTTP-ответом	278
10.3.1. Передача объектов в теле HTTP-ответа	279
10.3.2. Создание HTTP-ответа со статусом и заголовками.	281
10.3.3. Управление исключениями на уровне конечной точки . . .	284
10.4. Извлечение данных из тела запроса, полученного от клиента . . .	289
Резюме	292
Глава 11. Использование конечных точек REST.	293
11.1. Вызов конечной точки REST с помощью OpenFeign из Spring Cloud	297
11.2. Вызов конечных точек REST с помощью RestTemplate	301
11.3. Вызов конечной точки REST с помощью WebClient	305
Резюме	314
Глава 12. Использование источников данных в Spring-приложениях . . .	315
12.1. Что такое источник данных.	316
12.2. Взаимодействие с сохраненными данными с помощью JdbcTemplate.	320

12 Оглавление

12.3. Изменение конфигурации источника данных	330
12.3.1. Определение источника данных в файле свойств приложения	331
12.3.2. Использование нестандартного бина DataSource	333
Резюме	335
Глава 13. Транзакции в Spring-приложениях	337
13.1. Транзакции	340
13.2. Транзакции в Spring	340
13.3. Использование транзакций в Spring-приложениях	344
Резюме	356
Глава 14. Сохранение данных с помощью Spring Data	357
14.1. Что такое Spring Data	358
14.2. Как работает Spring Data	361
14.3. Использование Spring Data JDBC	367
Резюме	377
Глава 15. Тестирование Spring-приложений	379
15.1. Как писать правильные тесты	381
15.2. Реализация тестов в Spring-приложениях	384
15.2.1. Разработка модульных тестов	384
15.2.2. Разработка интеграционных тестов	398
Резюме	402
Приложение А. Архитектурные концепции	404
A.1. Монолитная архитектура	404
A.2. Сервис-ориентированная архитектура	408
A.2.1. Усложнение взаимодействия между сервисами	411
A.2.2. Усложнение обеспечения безопасности системы	412
A.2.3. Усложнение хранения данных	412
A.2.4. Усложнение развертывания системы	414
A.3. От микросервисов до бессерверных приложений	415
A.4. Что еще почитать	416
Приложение Б. Использование XML в конфигурации контекста	418

Приложение В. Краткое введение в HTTP	420
B.1. Что такое HTTP	420
B.2. HTTP-запросы как язык общения между клиентом и сервером	422
B.3. HTTP-ответ: способ получить ответ от сервера.	425
B.4. HTTP-сессия	428
Приложение Г. Представление данных в формате JSON	431
Приложение Д. Установка MySQL и создание базы данных	434
Шаг 1. Установка СУБД на локальном компьютере	435
Шаг 2. Установка клиентского приложения для выбранной СУБД.	435
Шаг 3. Установка соединения с локальной СУБД	435
Шаг 4. Создание базы данных	438
Приложение Е. Рекомендованные инструменты	442
Приложение Ж. Материалы, рекомендуемые для дальнейшего изучения Spring	444

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

Предисловие

Появившись в начале 2000-х годов, фреймворк Spring быстро превзошел своего конкурента EJB по простоте модели программирования, разнообразию функций и интегрированным сторонним библиотекам. Со временем Spring превратился в наиболее обширный и зрелый из существующих фреймворков для разработки, доступных на каком-либо языке программирования. Главный конкурент Spring сошел с дистанции, когда компания Oracle прекратила работу над платформой Java EE 8 и сообщество взяло ее поддержку на себя, переименовав в Jakarta EE.

Согласно последним обзорам, фреймворк Spring является основой более половины Java-приложений. Это огромная кодовая база, поэтому изучить Spring критически важно, ведь в своем профессиональном развитии вы неизбежно столкнетесь с данной технологией. Я занимался созданием приложений на основе Spring в течение 15 лет, и сегодня практически все обучаемые мной разработчики из сотен компаний используют эту экосистему.

Реальность такова, что, несмотря на всю популярность Spring, довольно трудно найти качественные вводные материалы по данному фреймворку. Его справочная документация насчитывает несколько тысяч страниц. Там описываются всевозможные тонкости и детали, которые могут пригодиться в очень специфических сценариях, но начинающему разработчику нужно не это. Онлайновые видеокурсы и учебники не особенно привлекательны для изучающих фреймворк, а в немногих книгах по основам Spring зачастую уделяется слишком много места обсуждению тем, которые, как оказалось, не имеют отношения к современным проблемам разработки приложений. Однако в книге Лауренсиу Спилкэ вы едва ли найдете что-нибудь лишнее: все изложенные здесь концепции постоянно встречаются в разработке любых Spring-приложений.

Данное издание аккуратно выведет вас на уровень, достаточный для быстрого достижения успеха в проекте, основанном на фреймворке Spring. Мой собственный опыт подготовки тысяч сотрудников показывает, что сейчас подавляющее большинство программистов, работающих на Spring, не понимают суть фреймворка так четко, как ее объясняет данная книга. Более того, разработчикам неизвестны многие подводные камни, о которых предупреждает книга. Считаю, что она обязательна к прочтению для любого разработчика, начинающего свой первый проект на Spring.

Та предусмотрительность, с которой Лауренциу предугадывает возможные вопросы читателей, доказывает его обширный опыт в преподавании Spring. Автор обращается к читателю тепло, по-дружески, благодаря чему изучать книгу легко и приятно. Издание имеет ясную, четкую структуру: я восхищаюсь тем, как сложные темы постепенно раскрываются и объясняются в основном материале, а затем повторяются в последующих главах.

Книга прекрасна также тем, что рассказывает читателю о фундаментальных проблемах преемственности проектов, в которых используется фреймворк Spring. Я обнаружил, что в экосистеме, построенной на Spring Boot, бывает очень полезно заглянуть за кулисы. А еще автор знакомит читателя с такими технологиями последнего поколения, как Feign-клиенты и даже реактивное программирование.

Желаю вам приятного чтения. Никогда не бойтесь закопаться в код, который кажется вам слишком сложным!

*Виктор Рента,
Java-рыцарь, тренер и консультант*

Введение

Делиться знаниями и создавать учебные материалы — мое хобби. Я не только разработчик программного обеспечения, но и преподаватель. С 2009 года я обучил Java тысячи разработчиков с разным уровнем опыта, от студентов университета до маститых специалистов из крупных корпораций. За последние несколько лет я пришел к выводу, что новички обязательно должны освоить Spring. Современные приложения больше не пишутся на чистых языках программирования — почти все они основаны на фреймворках. А поскольку в настоящее время Spring является самым популярным фреймворком для написания Java-приложений, именно с ним разработчику стоит познакомиться в первую очередь.

Преподавая Spring начинающим, я столкнулся с тем, что его все еще считают фреймворком, который следует изучать, уже имея некоторый опыт программирования. Есть множество учебных материалов, книг и статей на эту тему, однако мои студенты постоянно жаловались, что все эти материалы слишком сложны для них. Я понял, что проблема не в том, что существующие пособия недостаточно хороши, а в том, что среди них нет учебников для изучающих Spring с нуля. Поэтому я решил написать книгу, в которой не предполагается, что вы знакомитесь со Spring, уже имея некоторый опыт, — напротив, она рассчитана на читателя с минимальными фундаментальными знаниями.

Технологии меняются быстро. Но меняются не только они. Необходимо также улучшать способы обучения этим технологиям. Несколько лет назад можно было изучить основы языка программирования и стать разработчиком, не имея никакого представления о фреймворках. Однако сейчас ситуация изменилась. Овладевание лишь тонкостями языка программирования больше не гарантия того, что вы быстро получите навыки, необходимые для работы в команде программистов. Сегодня я рекомендую разработчикам, освоив начальную базу,

переходить к изучению одного из фреймворков для построения приложений. Spring, по моему мнению, — наилучший вариант, с которого следует начать. Понимание основ Spring откроет вам двери к другим технологиям. Таким образом вместо устаревшего линейного обучения вы получите нечто похожее на дерево, где каждая ветвь — это новый фреймворк, который вы будете осваивать параллельно с другими.

Я структурировал книгу так, чтобы вы захотели начать обучение Spring с ней. Постепенно, шаг за шагом, издание снабдит вас всеми необходимыми теоретическими знаниями и примерами, в которых описываемые темы будут продемонстрированы на практике. Надеюсь, данная книга принесет вам пользу, поможет быстро получить знания по Spring и откроет двери к дальнейшему изучению фреймворков.

Благодарности

Эта книга не появилась бы, если бы мне не помогали многие умные, профессиональные и доброжелательные люди.

Прежде всего, огромное спасибо моей жене Даниэле, которая всегда была рядом. Ее ценные замечания, бесконечная поддержка и ободрение невероятно мне помогли.

Также хочу выразить признательность и особо поблагодарить всех коллег и друзей, которые помогали мне цennыми советами, начиная с самого первого списка глав и заявки в редакцию.

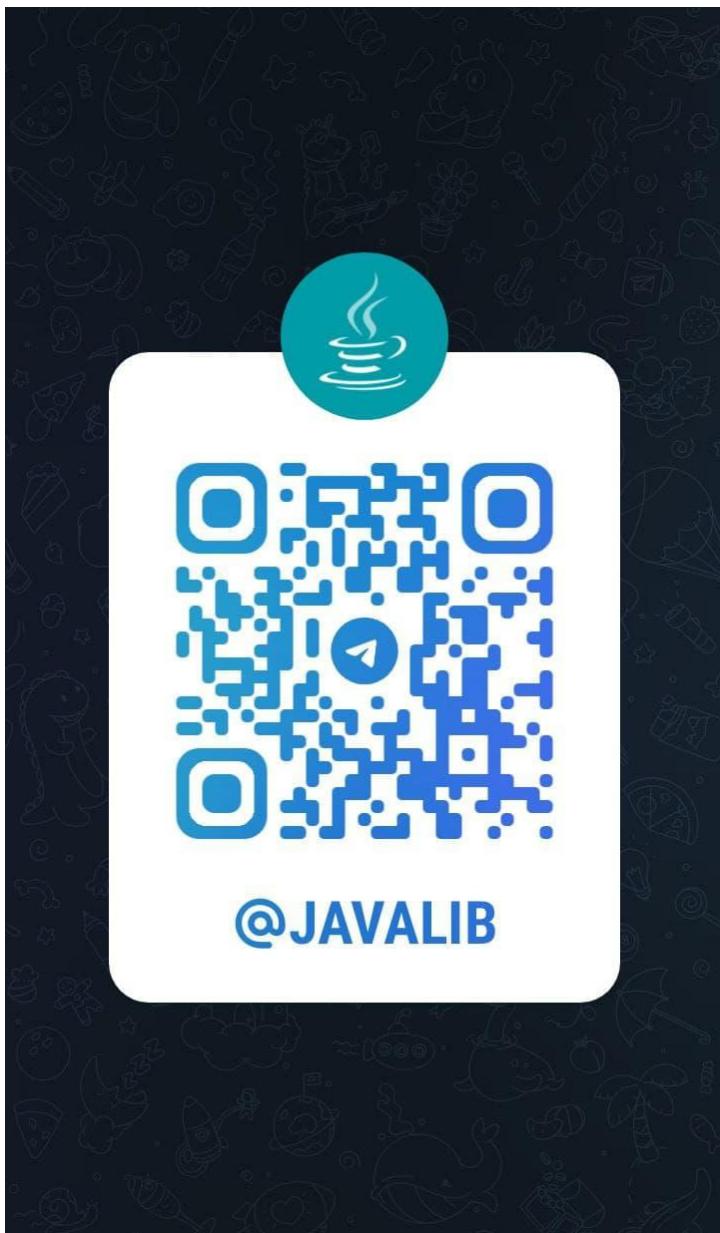
Большое спасибо всей команде издательского дома Manning за помощь в претворении моих идей в живую книгу. Особенно я признателен Марине Майлс, Алу Шереру и Джин-Франсуазе Морин за невероятную поддержку и професионализм. Ваши советы оказали огромное влияние на мою книгу.

Хочу поблагодарить мою подругу Иоану Гез за рисунки, которые она сделала для этой книги. Она превратила мои мысли в забавные картинки.

Хочу также выразить признательность всем рецензентам, которые на каждом этапе создания книги оставляли невероятно полезные отзывы: Аллену Ломпо, Александру Карпенко, Андреа-Карло Гранате, Андреа Пачолле, Андре-Дамиану Сакко, Эндрю Освальду, Бобби Лину, Бонни Малец, Кристиану Крейцер-Беку, Дэниэлу Карлу, Дэвиду-Лайелу Орпену, Деандре Рашону, Харинату Кунтамук-кале, Ховарду Уоллу, Жерому Батону, Джиму Уэлшу, Жуану-Мигелю-Пиресу

Диасу, Лучиану Энаке, Мэтту Д., Мэтью Грину, Микаэлю Бистрему, Младену Кнежичу, Натану Б. Крокеру, Пьеру-Мишелью Анселью, Раджешу Моханану, Рикардо Ди Паскуале, Суните Чоудхури, Тану Ви и Зохебу Айнапуре. Благодаря вам эта книга стала гораздо лучше.

Наконец, особая благодарность моим друзьям — Марии Китцу, Андреа Тудосе, Флорин Чукулеску и Даниэле Илеана за советы, которые они давали мне все время, пока я писал эту книгу.



О книге

Раз уж вы открыли эту книгу, предположу, что вы являетесь разработчиком программного обеспечения в экосистеме Java и решили, что вам стоит познакомиться со Spring. С помощью данного издания вы изучите основы Spring, даже если до сих пор ничего не знали о фреймворках в целом и, разумеется, о Spring в частности.

Сначала вы узнаете, что вообще такое фреймворк, а затем постепенно на соответствующих примерах изучите основы Spring. Вы не только научитесь использовать компоненты и возможности фреймворка, но и познакомитесь с основными принципами, на базе которых эти возможности реализованы. Понимание того, что делает фреймворк, когда вы применяете тот или иной компонент, поможет вам проектировать более качественные приложения и быстрее справляться с проблемами.

Когда вы дочитаете это издание, вы будете обладать такими крайне важными при разработке приложений навыками, как:

- настройка и использование контекста Spring и внедрение зависимостей в Spring;
- разработка и использование аспектов;
- реализация веб-приложений;
- реализация обмена данными между приложениями;
- сохранение данных;
- тестирование реализаций.

Эта книга будет полезной:

- в повседневной работе с приложениями, использующими Spring;
- для успешного прохождения технического собеседования на позицию Java-разработчика;
- для получения сертификата по Spring.

И хотя подготовить вас к сертификации по Spring не является главной целью настоящего издания, все же полагаю, что его обязательно нужно прочесть, прежде чем вы углубитесь в детали, которые обычно требуются для сертификационного экзамена.

КТО ДОЛЖЕН ПРОЧИТАТЬ ЭТУ КНИГУ

Книга предназначена для тех разработчиков, которые владеют основами объектно-ориентированного программирования и концепциями Java и хотят изучить Spring с нуля либо освежить базовые знания по нему. От вас не требуется знакомство с каким-либо фреймворком, но нужно знать основы Java, поскольку именно этот язык используется в примерах, приводимых в книге.

Spring является одной из самых распространенных технологий, применяемых в Java-приложениях. В будущем, скорее всего, данный фреймворк станет использоваться еще интенсивнее. Поэтому современный Java-разработчик просто обязан изучить Spring. Освоив то, чему я собираюсь научить вас в этой книге, вы повысите свою квалификацию, получите понимание основ Spring, овладеете навыками, необходимыми для успешного прохождения интервью на позицию Java-разработчика, а также сможете создавать приложения с использованием технологий Spring. Это издание также откроет для вас возможность дальнейшего изучения более сложных нюансов Spring.

СТРУКТУРА ИЗДАНИЯ

Данная книга состоит из 15 глав, разделенных на две части. В первой части мы начнем обсуждение с простых примеров, где я покажу вам, как сообщить Spring о существовании вашего приложения. Затем мы выполним несколько примеров, чтобы вам стало понятно, на чем базируется любое реальное Spring-приложение. Закончив с основами Spring Core, мы перейдем к обсуждению Spring Data и Spring Boot.

Начиная с главы 2, теоретические вопросы будут сопровождаться примерами, в которых мы будем применять полученные знания. Я буду объяснять код этих примеров фрагмент за фрагментом. Советую выполнять их по мере чтения книги — так вы сможете сравнить свои результаты с тем, что получилось у меня.

Как показано на рис. 1, я выстроил главы в определенном порядке. В главах с 2-й по 5-ю, где обсуждается контекст Spring, вы найдете преимущественно теоретические примеры. Мало знакомому или вовсе не знакомому со Spring читателю необходимо начинать именно с них. Не беспокойтесь, я опишу основы самым простым из возможных способов. Затем наши примеры и обсуждения будут постепенно усложняться и начнут соответствовать реальному коду промышленного качества.

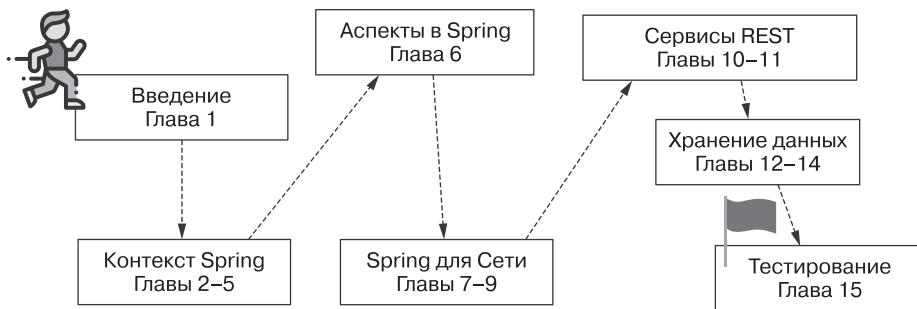


Рис. 1. Если вы ничего или почти ничего не знаете о Spring, то лучше всего при изучении данной книги начать с первой главы и дальше читать все подряд

Если вы уже хорошо знакомы с контекстом Spring и Spring AOP, можете пропустить часть I и сразу перейти к части II – «Реализация» (главы 7–15), как показано на рис. 2.

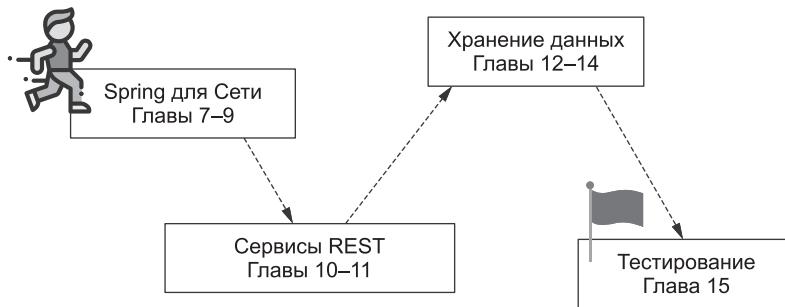


Рис. 2. Если вы уже знакомы с основами фреймворка Spring, умеете использовать контекст Spring и знаете аспекты проектирования, можете начать с части II, где мы применим возможности Spring для построения сценариев зеркального отображения приложений, с которыми вы столкнетесь в реальных системах

К тому моменту, как вы дочитаете настоящую книгу, вы освоите множество навыков профессиональной разработки приложений. Вы узнаете, как, используя наиболее распространенные современные технологии, подключиться к базе данных

и как организовать коммуникацию между приложениями. В конце издания мы рассмотрим критически важную тему: тестирование. Я буду постоянно вставлять в текст истории из собственной практики и заметки с полезными советами.

Следует помнить, что Spring — это целая вселенная. Так что, прочитав только одну книгу, вы не узнаете всего. Данное издание станет для вас лишь началом знакомства с этим фреймворком; благодаря ему вы освоите фундаментальные навыки использования важнейших компонентов Spring. На протяжении всей книги, там, где это уместно, я буду давать ссылки на другие ресурсы и книги, в которых рассматриваемые темы описываются более подробно. Настоятельно рекомендую вам ознакомиться с этими дополнительными ресурсами и книгами.

О КОДЕ

В этой книге представлено около 70 проектов, над которыми мы будем работать в главах 2–14. Разбирая конкретный пример, я буду указывать название проекта, в котором этот пример реализован. Советую писать код самостоятельно и использовать проекты в книге только для сравнения ваших решений с моими. Такой подход позволит вам глубже понять изучаемые концепции.

Примеры проектов, рассмотренных в книге, можно скачать по адресу <https://manning-content.s3.amazonaws.com/download/a/32357a2-2420-4c0f-be67-645246ae0d94/code.zip>.

Все рассмотренные проекты собирались с помощью Maven, чтобы упростить их импорт в любую IDE. Я писал проекты в IntelliJ IDEA, но вы можете запускать их из Eclipse, Netbeans или любой другой среды, на ваш выбор. Обзор рекомендованных инструментов представлен в приложении Е.

В данной книге есть много примеров исходного кода, как в виде пронумерованных листингов, так и прямо в тексте. В обоих случаях исходный код оформляется таким монотипным шрифтом, чтобы можно было отличить его от остального текста. Иногда используются **выделения жирным шрифтом**, чтобы показать, что код изменился по сравнению с предыдущими этапами (например, когда в существующую строку добавлялся новый функционал). Во многих случаях первоначальный вид кода был изменен — вставлены разрывы строк и изменены отступы, чтобы код поместился по ширине книжной страницы. Изредка и этого было недостаточно, и тогда в листинги вставлялись символы продолжения строки (**►**). Кроме того, если код описан в тексте, то из него часто удалялись комментарии. Многие листинги сопровождаются аннотациями, в которых подчеркиваются основные концепции.

Об авторе



Лауренциу Спилкэ — глава отдела разработки и тренер в компании Endava, где он руководит разработкой проектов для финансового рынка Европы, США и Азии. Его опыт работы насчитывает более десяти лет. По мнению Лауренциу, важно не только создавать высококачественное программное обеспечение, но также делиться знаниями и помогать другим повышать свою квалификацию. Руководствуясь этими соображениями, Лауренциу создал курсы по технологиям Java и сам преподает там, проводя презентации и мастер-классы. Его «Твиттер» — @laurspilca.

Иллюстрация на обложке

Рисунок на обложке книги называется «Женщина из Аяччо на Корсике». Это иллюстрация из коллекции «Костюмы народов мира» Жака Грассе де Сен-Совера (1757–1810), изданной во Франции в 1797 году. Каждая иллюстрация тщательно прорисована и раскрашена вручную. Богатство коллекции Грассе де Сен-Совера живо напоминает нам о том, как далеки были друг от друга культуры разных городов и регионов всего 200 лет назад. Изолированные, люди говорили на разных диалектах и языках. Встретив человека на городской улице или в сельской местности, мы бы легко поняли, откуда он родом, чем занимается и каков его достаток, всего лишь взглянув на его одежду.

С тех пор наша манера одеваться изменилась. Сегодня региональные различия, столь явные в прошлом, стерлись. Сейчас вы едва ли определите по одежде жителей разных континентов, не говоря уже о городах, регионах и странах. Похоже на то, что мы, поступившись культурным разнообразием, получили взамен более многогранную жизнь — разумеется, это касается в том числе и более многогранной и быстро меняющейся жизни технологий.

Сейчас, когда бывает трудно внешне отличить одну книгу о компьютерах от другой, мы поощряем изобретательность и инициативу компьютерного бизнеса, размещая на обложках своих книг рисунки Грассе де Сен-Совера и показывая, сколь сильно различалась жизнь в разных странах всего пару сотен лет назад.

От издательства

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

Мы будем рады узнать ваше мнение!

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

Часть I

Основные принципы

У любого строения есть фундамент — и фреймворки не исключение. Изучив часть I, вы научитесь использовать базовые компоненты, обеспечивающие работу фреймворка Spring, — контекст и аспекты. Именно на эти важнейшие компоненты опираются все функции Spring.

1

Spring в реальном мире

В этой главе

- ✓ Что такое фреймворк.
- ✓ Когда нужно, а когда не следует использовать фреймворки.
- ✓ Что такое фреймворк Spring.
- ✓ Как применять Spring в реальных условиях.

Фреймворк Spring (или просто Spring) — это прикладной фреймворк, который является частью экосистемы Java. *Прикладной фреймворк* — это пакет типичных функций программного обеспечения, образующих базовую структуру для разработки приложения. Прикладной фреймворк позволяет тратить меньше усилий при написании приложения, так как не приходится создавать весь код программы с нуля.

В настоящее время Spring используется для написания самых разных программ, от крупных серверных решений до средств автоматизации тестирования. Согласно многочисленным отчетам об исследованиях в области Java-технологий, таких как JRebel 2020 года или JAXEnter, Spring в настоящее время является наиболее востребованным фреймворком.

Spring популярен: разработчики стали чаще использовать его не только с Java, но и с другими JVM-языками. В последние несколько лет наблюдается впечатляющее увеличение числа разработчиков, применяющих Spring с Kotlin (еще одним популярным языком семейства JVM). В этой книге мы сосредоточимся на основах Spring: я познакомлю вас с важнейшими аспектами его использования

на примерах реальных приложений. Чтобы вам было удобнее и вы могли лучше сфокусироваться на Spring, я буду использовать только примеры на Java. В издании вы изучите и отработаете на практике такие основные приемы, как подключение к базе данных, установка соединения между приложениями, обеспечение безопасности и тестирование приложений.

Прежде чем перейти к следующим главам и углубиться в технические детали, поговорим о фреймворке Spring в целом и о том, где его стоит применять. Почему Spring столь популярен и когда им следует пользоваться?

В данной главе на примере Spring мы подробно рассмотрим, что такое фреймворк вообще. В разделе 1.1 мы обсудим преимущества использования фреймворков. В разделе 1.2 вы познакомитесь с экосистемой Spring и с теми ее компонентами, которые понадобятся для начала работы с ней. Затем я расскажу вам о возможных областях применения фреймворка Spring и о реальных сценариях из практики, раскрытых, в частности, в разделе 1.3. В разделе 1.4 мы обсудим случаи, когда использование фреймворка может оказаться неудачным решением. Все эти моменты следует прояснить прежде, чем вы начнете использовать Spring, — чтобы потом не забивать гвозди микроскопом.

Возможно, в зависимости от ваших исходных знаний, эта глава покажется вам сложной. В ней представлены понятия, с которыми вы, вероятно, еще не знакомы — и это может вызывать беспокойство. Но не волнуйтесь — даже если вы пока какие-то моменты не поймете, в процессе изучения книги они прояснятся. Иногда в процессе изложения я буду ссылаться на что-то описанное в предыдущих главах. Я буду так делать, поскольку изучение такого фреймворка, как Spring, не проходит линейно. Иногда придется подождать, пока соберется еще несколько кусочков пазла и станет видна вся картина. Но в итоге вы получите ясное представление о Spring и приобретете ценные навыки, необходимые для профессиональной разработки приложений.

1.1. ЗАЧЕМ НУЖНЫ ФРЕЙМВОРКИ

Чтобы у вас появилось желание что-то использовать, вам нужно знать, чем это «что-то» может быть вам полезно. Это касается и Spring. В данном разделе мы поговорим о фреймворках. Что это такое? Как и зачем появилась концепция фреймворков? Я познакомлю вас с этими важными моментами, для чего поделюсь собственными накопленными знаниями и покажу, как использовать различные фреймворки, включая Spring, при решении практических задач.

Прикладной фреймворк — это набор функционала, на базе которого строятся приложения. Прикладной фреймворк предоставляет широкий набор инструментов и функций, которые можно применять в разработке. Вы не обязаны использовать их все. В зависимости от требований, предъявляемых к разрабатываемому приложению, вы будете выбирать ту часть фреймворка, которая вам нужна.

В отношении прикладных фреймворков мне нравится такая аналогия. Приходилось ли вам покупать мебель, скажем, в «Икее»? Предположим, вы решили приобрести платяной шкаф. Но вы получите не готовый шкаф, а набор деталей для его сборки и инструкцию по сборке. А теперь представьте, что вы заказали шкаф, но вместо необходимых частей вам выдали вообще все, из чего можно собрать любую мебель: стол, шкаф и т. п. Если вам нужен только шкаф, то придется найти в этой куче нужное и собрать его. Именно так работает прикладной фреймворк: он предлагает все возможные компоненты программного обеспечения, которые могут понадобиться при создании любого приложения. Вам необходимо знать, какие функции выбрать и как их собрать, чтобы получить правильный результат (рис. 1.1).



Рис. 1.1. Дэвид заказал шкаф в магазине «Сделай сам». Но магазин (фреймворк) доставил Дэвиду (программисту) не только те детали (компоненты программного обеспечения), которые нужны для сборки его шкафа (приложения), а вообще все возможные детали, которые могут понадобиться при сборке любого шкафа. И Дэвиду (программисту) придется самому решать, какие детали (программные компоненты) необходимы и как их собрать, чтобы получить желаемый результат (приложение)

Концепция фреймворка не нова. На протяжении всей истории разработки ПО программисты замечали, что многие фрагменты написанного ими кода можно использовать повторно в других приложениях. Поначалу, когда приложений было мало, каждое из них было уникально и писалось с нуля на том или ином языке программирования. По мере того как область применения программного

обеспечения расширялась и на рынке появлялось все больше продуктов, стало еще заметнее, как много приложений имеют сходные требования. Вот некоторые из них:

- в каждом приложении предусмотрены уведомления при ошибках входа, предупреждения и информационные сообщения;
- в большинстве приложений для обработки изменений данных используются транзакции. Транзакции являются важным механизмом, который обеспечивает целостность данных. Подробнее мы рассмотрим эту тему в главе 13;
- в большинстве приложений применяются механизмы защиты от одних и тех же распространенных уязвимостей;
- в большинстве приложений используются одни и те же механизмы обмена данными с другими приложениями;
- в большинстве приложений есть одни и те же механизмы повышения производительности, такие как кэширование и сжатие данных.

Данный список можно продолжать. В сущности, код бизнес-логики, реализованной в приложении, значительно меньше, чем вся эта внутренняя начинка (которую еще называют «сантехникой»).

Говоря о «коде бизнес-логики», я имею в виду код, реализующий бизнес-требования приложения. Именно он реагирует на ожидания пользователя от продукта. Например, пользователю нужно, чтобы при щелчке кнопкой мыши на определенной ссылке был сформирован отчет. Этую функциональность реализует часть кода создаваемого вами приложения — и именно эту часть разработчики называют кодом бизнес-логики. Однако любое приложение делает и ряд других вещей: обеспечивает безопасность, журналирование, целостность данных и т. п. (рис. 1.2).

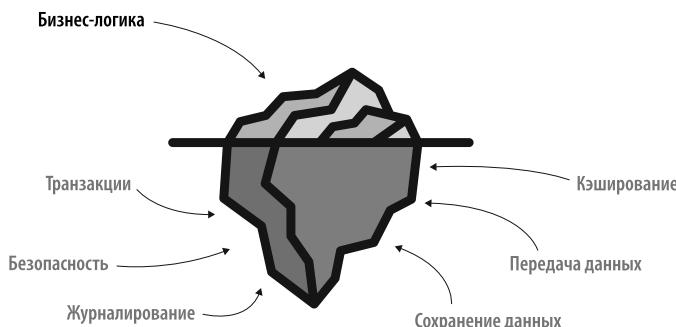


Рис. 1.2. Взгляд пользователю доступна только верхушка айсберга. Пользователи видят главным образом результат работы бизнес-логики. Но это лишь малая доля того, что представляет собой полный функционал приложения. Подобно айсбергу, большая часть которого находится под водой и скрыта от глаз, мы не видим основную часть кода корпоративного приложения, поскольку она выполняется посредством зависимостей

Более того, именно бизнес-логика отличает одно приложение от другого с точки зрения функционала.

Если взять два приложения разных типов — например, систему поиска попутчиков и социальную сеть, — то увидим, что сценарии использования у них разные.

ПРИМЕЧАНИЕ

Сценарий использования (use case) — это причина, по которой человек использует приложение. Например, в приложении поиска попутчиков сценарием использования будет «найти машину». А для приложения доставки еды — «заказать пиццу».

Вы можете выполнять различные действия, но в обоих случаях они будут требовать сохранения данных, их передачи, журнализации, настройки параметров безопасности, возможно, кеширования и т. п. Функционал, не относящийся к бизнес-логике, может многократно внедряться в разные приложения. Имеет ли смысл каждый раз заново его переписывать? Разумеется, нет:

- ведь, используя готовые модули вместо их разработки с нуля, вы не тратите время и деньги зря;
- в готовом решении, применяемом во многих приложениях, вероятность ошибок меньше, так как его уже протестировали другие люди;
- поскольку многие разработчики уже освоили этот функционал, вы можете пользоваться советами сообщества. Если бы вы реализовали собственный код, в нем разбиралось бы гораздо меньше людей.

ИСТОРИЯ ПЕРЕХОДА

Одним из первых продуктов, над которыми я работал, была огромная система, написанная на Java. Она состояла из нескольких приложений, построенных на базе сервера с устаревшей архитектурой. Все они были написаны с нуля на Java SE. Разработка приложения на данном языке началась 25 лет назад — главным образом поэтому оно было таким, каким было. На момент его создания никто не мог представить, до каких размеров оно однажды дорастет. Тогда еще не существовало более совершенных концепций системной архитектуры; из-за низкой скорости интернет-соединений все, как правило, реализовывалось отдельно в рамках каждой системы и работало по-разному.

Но время шло — и спустя годы приложение стало похоже на большой ком грязи. По ряду уважительных причин (которые я не буду здесь упоминать) команда разработчиков решила, что необходимо перейти на более современную архитектуру. Эти изменения подразумевали в первую очередь очистку кода. И одним из первых шагов было использование фреймворка. Мы выбрали Spring. На то время у нас был Java EE (сейчас это Jakarta EE), но большинство участников проекта решили, что лучше перейти на Spring, так

как этот фреймворк является более простой альтернативой, которую легче реализовать и проще поддерживать.

Переход был не из простых. Вместе с парой коллег, каждого из которых был экспертом в своей области, а также хорошо разбирался в работе самого приложения, мы потратили на это преобразование много сил.

Но результат был потрясающим! Мы удалили более 40 % строк кода. Именно в этот момент я впервые понял, какое серьезное влияние может оказывать использование фреймворка.

ПРИМЕЧАНИЕ

Выбор и использование фреймворка зависит от конструкции и архитектуры приложения. Изучая фреймворк Spring, вы поймете, что имеет смысл исследовать также и эти аспекты. В приложении А вы найдете описание архитектур программного обеспечения и ссылки на отличные ресурсы на тот случай, если захотите углубиться в данную тему.

1.2. ЭКОСИСТЕМА SPRING

Рассмотрим Spring и связанные с ним проекты Spring Boot и Spring Data. В книге вы узнаете о них все и вдобавок получите полезные ссылки. На практике часто используют несколько фреймворков одновременно, каждый из которых помогает ускорить разработку определенной части приложения.

Мы называем Spring фреймворком, но в действительности он гораздо сложнее. Spring — это целая экосистема фреймворков. Как правило, когда разработчики упоминают фреймворк Spring, они имеют в виду часть программного функционала, которая включает в себя следующее.

1. **Spring Core** — фундаментальная часть Spring, в которой реализован его базовый функционал. Одной из этих функций является контекст Spring. Как будет подробно описано в главе 2, контекст Spring — это фундаментальная функциональная возможность, благодаря которой Spring может управлять экземплярами приложения. Также частью функционала Spring Core являются аспекты Spring. С ними Spring может перехватывать определенные в приложении методы и манипулировать ими (мы подробно рассмотрим аспекты в главе 6). Еще один компонент, который вы обнаружите в составе Spring Core, — это Spring Expression Language (SpEL). Он позволяет описывать конфигурации Spring с помощью специального языка. Наверняка для вас все это новые понятия — я не ожидаю, что вам приходилось слышать о них ранее. Но вскоре вы поймете, что в состав Spring Core входят механизмы, позволяющие интегрировать Spring в ваше приложение.

2. **Spring MVC (model-view-controller, «модель – представление – контроллер»).** Эта часть фреймворка Spring позволяет создавать веб-приложения, обрабатывающие HTTP-запросы. Мы будем использовать Spring MVC, начиная с главы 7.
3. **Spring Data Access** — еще одна базовая часть Spring. Она предоставляет основные инструменты для соединения с базами данных SQL, что позволяет реализовать уровень доступа к данным в приложении. Мы будем использовать Spring Data Access, начиная с главы 13.
4. **Spring Testing.** Эта часть фреймворка включает в себя инструменты, позволяющие писать тесты для Spring-приложения. Мы рассмотрим эту тему в главе 15.

Для начала вы можете представить фреймворк Spring в виде планетной системы, в которой Spring Core играет роль центральной звезды и притягивает к себе остальные части фреймворка (рис. 1.3).

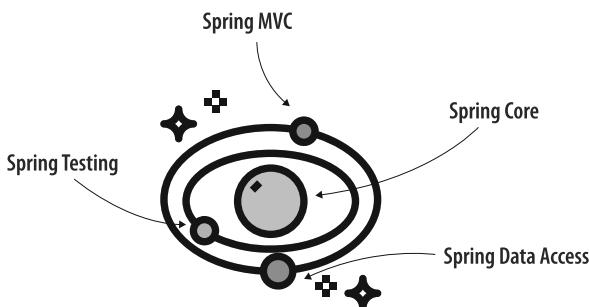


Рис. 1.3. Фреймворк Spring можно представить в виде планетной системы, в центре которой находится Spring Core. Модули программного обеспечения играют роль планет, которые врачаются вокруг «звезды» Spring Core и удерживаются ее гравитационным полем

1.2.1. Spring Core вблизи: ядро Spring

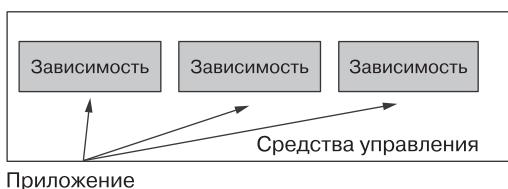
Spring Core — это та часть фреймворка Spring, которая обеспечивает фундаментальные механизмы его интеграции в приложение. Spring работает по принципу *инверсии управления* (inversion of control, IoC): вместо того чтобы приложение само контролировало свое выполнение, управление передается некоторому другому программному обеспечению — в данном случае фреймворку Spring. Посредством системы настроек мы предоставляем фреймворку инструкции о том, как распоряжаться написанным нами кодом, что и определяет логику работы приложения. Именно это и подразумевается под «инверсией» в аббревиатуре IoC: мы не позволяем приложению управлять собственным выполнением

посредством его же кода или использовать зависимости. Вместо этого мы передаем фреймворку (зависимости) управление приложением и его кодом (рис. 1.4).

ПРИМЕЧАНИЕ

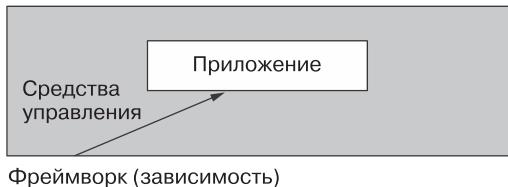
В этом контексте «управление» означает такие действия, как «создание экземпляра» или «вызов метода». Фреймворк может создавать объекты классов, определенных в приложении. На основании написанных вами конфигураций Spring способен перехватывать выполнение метода и дополнять его различными функциями, к примеру регистрацией любой ошибки, которая может возникнуть в процессе выполнения этого метода.

Без IoC



Приложение выполняет само себя
и управляет зависимостями
(использует их) по мере необходимости

С IoC



Приложение выполняется
под управлением фреймворка
(зависимости)

Рис. 1.4. Инверсия управления: вместо того чтобы выполнять собственный код, в котором используется несколько зависимостей, в сценарии IoC приложение выполняется под управлением зависимости. В данном случае выполнением приложения управляет фреймворк Spring. Таким образом, Spring реализует сценарий выполнения IoC

Мы начнем изучение Spring со Spring Core, а изучение Spring Core — с функционала Spring IoC, который рассмотрим в главах 2–5. Контейнер IoC «склеивает» компоненты Spring между собой, а компоненты приложения — с фреймворком. Благодаря контейнеру IoC, который часто называют контекстом Spring, объекты становятся видимыми для Spring, и фреймворк может их использовать в соответствии с заданной вами конфигурацией.

В главе 6 мы продолжим рассматривать аспектно-ориентированное программирование (aspect-oriented programming, AOP) в Spring. Spring позволяет управлять

экземплярами, помещенными в контроллер IoC, и, в частности, перехватывать методы, описывающие поведение этих экземпляров. Такая возможность называется *аспектированием* метода. Spring AOP — один из наиболее типичных способов взаимодействия фреймворка с приложением. Это свойство также делает Spring AOP одной из необходимейших частей фреймворка. В состав Spring Core входят также управление ресурсами, интернационализация (i18n), преобразование типов и SpEL. Все эти функции неоднократно встречаются нам в примерах на протяжении всей книги.

1.2.2. Сохранение данных приложения с помощью Spring Data Access

Для большинства приложений критически важно сохранять часть обрабатываемых данных. Работа с базами данных — фундаментальная часть проекта. В Spring для сохранения данных, как правило, применяется модуль Data Access. В нем используется, в частности, управление транзакциями, JDBC и интеграция с фреймворками, реализующими *объектно-реляционную привязку* (object-relational mapping, ORM), например, с Hibernate (если вы еще не знаете, что такое ORM-фреймворк, или ничего не слышали о Hibernate — ничего страшного; мы рассмотрим эти аспекты далее в книге). В главах 12–14 мы изучим все необходимое, чтобы вы смогли начать работать со Spring Data Access.

1.2.3. Возможности Spring MVC для разработки веб-приложений

Большинство приложений, разрабатываемых на Spring, — это веб-приложения. Поэтому в экосистеме Spring вы обнаружите большой набор инструментов, позволяющих создавать всевозможные веб-приложения и веб-сервисы. С помощью Spring MVC можно писать их в виде обычных сервлетов, что, как правило, и делают для огромного количества современных продуктов. Мы рассмотрим использование Spring MVC более подробно в главе 7.

1.2.4. Тестирование в Spring

Модуль тестирования Spring включает в себя большой набор инструментов, которые мы будем использовать для написания модульных и интеграционных тестов. О тестировании написано немало страниц, но мы в главе 15 рассмотрим лишь то, что нужно для начала тестирования в Spring. Я также дам ссылки на ряд полезных ресурсов, которые стоит изучить, чтобы более детально разобраться в теме. Как показывает мой опыт, нельзя стать настоящим разработчиком, не освоив тестирования, так что вам необходимо в это вникнуть.

1.2.5. Проекты на базе экосистемы Spring

Экосистема Spring – это гораздо больше, чем просто функции, описанные выше. Она включает в себя множество других фреймворков, хорошо интегрированных между собой и образующих целую вселенную. Здесь вы найдете такие проекты, как Spring Data, Spring Security, Spring Cloud, Spring Batch, Spring Boot и т. д. При разработке вы можете использовать сразу несколько из них. Например, можно построить приложение на возможностях Spring Boot, Spring Security и Spring Data. В следующих главах мы будем работать над небольшими заданиями, в которых будем применять функционал различных проектов экосистемы Spring. Под проектом я подразумеваю независимо разработанную часть экосистемы Spring. Каждый из них – плод работы отдельной команды, продолжающей расширять его возможности. Все проекты имеют отдельное описание и свою ссылку на официальном веб-сайте Spring: <https://Spring.io/projects>.

Но я буду упоминать Spring Data и Spring Boot и за пределами вселенной Spring. Эти проекты часто используются в приложениях, так что важно изучить их сразу.

Расширение возможностей хранения данных с помощью Spring Data

Проект Spring Data – это часть экосистемы Spring, которая позволяет легко подключаться к базам данных и использовать уровень доступа к данным, написав минимальное количество строк кода. Проект обращается как к SQL, так и к NoSQL-базам данных и имеет высокоуровневую обертку, которая упрощает работу с хранением данных.

ПРИМЕЧАНИЕ

В нашем распоряжении есть модуль Spring Data Access, который является частью Spring Core, и Spring Data, который является независимым проектом экосистемы Spring. Spring Data Access обеспечивает реализацию фундаментальных средств доступа к данным, таких как механизм транзакций и инструменты JDBC. Spring Data упрощает доступ к базам данных и предлагает расширенный инструментарий, который делает разработку более понятной и обеспечивает возможность подключения приложения к разнообразным источникам данных. Мы рассмотрим эту тему подробнее в главе 14.

Spring Boot

Проект Spring Boot – это часть экосистемы Spring, реализующая концепцию «соглашения важнее конфигурации». Главная идея этой концепции состоит в следующем. Вместо того чтобы описывать все параметры конфигурации в самом фреймворке, Spring Boot предлагает некую конфигурацию по умолчанию, которую можно изменять по мере необходимости. В результате, как правило, приходится писать меньше кода, поскольку вы следуете известным соглашениям и ваше приложение отличается от других лишь несколькими мелкими деталями.

Поэтому будет более эффективно не описывать заново всю конфигурацию в каждом новом приложении, а начать с некоторой конфигурации по умолчанию и изменить в ней лишь то, что отличается от общепринятого. Мы будем подробно рассматривать Spring Boot, начиная с главы 7.

Экосистема Spring огромна и включает в себя множество проектов. Некоторые из них будут встречаться чаще, чем другие; а некоторые, возможно, вам вообще не понадобятся, если только не придется разработать какое-то специфическое приложение. В этой книге будут упоминаться только те проекты, которые действительно необходимы на начальном этапе: Spring Core, Spring Data и Spring Boot. Полный список проектов экосистемы Spring вы найдете на официальном веб-сайте Spring: <https://Spring.io/projects/>.

АЛЬТЕРНАТИВЫ SPRING

Если углубиться в серьезное обсуждение альтернатив Spring, то некоторые читатели могут по ошибке посчитать их альтернативами всей экосистеме. Но аналоги есть у многих отдельных компонентов и проектов, входящих в экосистему Spring, — другие фреймворки и библиотеки, коммерческие или с открытым кодом.

Возьмем, к примеру, контейнер Spring IoC. Было время, когда разработчики очень ценили спецификацию Java EE. Обладая немного иной философией, Java EE (код которой в 2017 году был переработан, открыт и назван Jakarta EE — <https://jakarta.ee/>) предлагает такие спецификации, как Context and Dependency Injection (CDI) и Enterprise Java Beans (EJB). CDI и EJB можно использовать для управления контекстом экземпляров объектов и для реализации аспектов (которые в терминологии EE называются interceptors — «перехватчики»). Долгое время хорошим фреймворком для управления экземплярами объектов в контейнере был Google Guice (<https://github.com/google/guice>).

Для некоторых проектов удается найти одну или несколько альтернатив. Например, вместо Spring Security можно использовать Apache Shiro (<https://shiro.apache.org/>). Или же вместо Spring MVC и технологий Spring можно построить веб-приложение на фреймворке Play (<https://www.playframework.com/>).

Из более новых проектов многообещающе выглядит Red Hat Quarkus (<https://quarkus.io/>). Quarkus был создан для внедрения облачных решений и сейчас быстро совершенствуется. Я не удивлюсь, если однажды он станет одним из ведущих проектов для разработки промышленных приложений в экосистеме Java.

Мой вам совет: всегда рассматривайте альтернативы. При разработке программного обеспечения необходимо держать глаза открытыми и никогда не верить в одно решение, «единственное и неповторимое». Вам постоянно будут встречаться сценарии, в которых одна технология работает лучше, чем другая.

1.3. SPRING В РЕАЛЬНЫХ ЗАДАЧАХ

Теперь, когда вы получили общее представление о Spring, вы понимаете, когда и как следует использовать данный фреймворк. Далее я приведу несколько примеров приложений, для которых Spring подошел бы идеально. Мне слишком часто встречались разработчики, которые считали, что фреймворки наподобие Spring подходят лишь для серверных приложений. Более того, я наблюдал тенденцию, когда область применения Spring сузилась еще сильнее — до серверных веб-приложений. Действительно, во многих случаях Spring используется именно в таком ключе, однако важно помнить, что этим возможности фреймворка не ограничиваются. Мне встречались команды, успешно использующие Spring для разработки самых разных приложений, таких как средства автоматизации тестирования и даже автономные десктопные решения.

Далее я покажу вам несколько реальных задач, которые на моих глазах были успешно решены с применением Spring. С одной стороны, это далеко не все возможные сценарии; с другой стороны, и в этих случаях Spring не всегда будет работать хорошо. Вспомните, о чем мы говорили в разделе 1.2: фреймворк не всегда удачный выбор. Но для следующих общих случаев Spring, как правило, отлично подходит.

1. Разработка серверных приложений.
2. Разработка средств автоматизации тестирования.
3. Разработка десктопных приложений.
4. Разработка мобильных приложений.

1.3.1. Использование Spring для разработки серверных приложений

Серверное приложение — это часть системы, которая выполняется на стороне сервера и отвечает за управление данными и обслуживание запросов от клиентских приложений. Пользователи получают доступ к функционалу такого приложения через клиентские приложения, к которым пользователи, в свою очередь, обращаются напрямую. Далее клиентские приложения делают запросы серверному приложению для обработки данных пользователей. Серверное приложение может использовать базы данных для хранения информации или взаимодействовать различными способами с другими серверными приложениями.

В качестве примера вы легко можете вспомнить знакомое вам серверное приложение: это приложение, осуществляющее транзакции по банковским счетам. Пользователи получают доступ к своим счетам и управляют ими через мобильное или веб-приложение (интернет-банкинг). И мобильные, и веб-приложения являются клиентами серверного приложения. Для контроля пользовательских

транзакций серверное приложение должно взаимодействовать с другими серверными решениями, а часть данных, которыми оно управляет, должна сохраняться в базе данных. На рис. 1.5 представлена визуализация архитектуры такой системы.

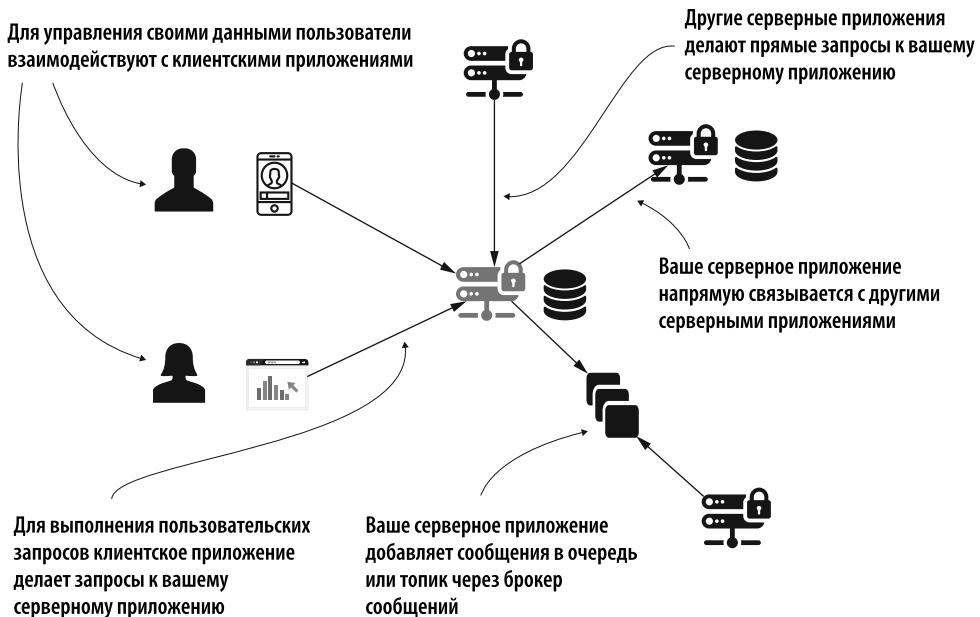


Рис. 1.5. Для управления данными серверное приложение различными способами взаимодействует с другими приложениями и подключается к базам данных. Как правило, серверные приложения достаточно сложны и требуют применения различных технологий. Фреймворки упрощают разработку серверных приложений, так как предоставляют инструментарий, позволяющий быстрее реализовать серверные решения

ПРИМЕЧАНИЕ

Не страшно, если вы пока не понимаете всех деталей, изображенных на рис. 1.5. Я не ожидаю от вас знаний о том, что такое брокер сообщений; вы даже можете не знать, как организовать обмен данными между компонентами. Я лишь хочу, чтобы вы увидели, что в реальном мире такие системы бывают сложными, и поняли, что проекты экосистемы Spring созданы, чтобы позволить избежать этой сложности, насколько возможно.

Spring предлагает разработчику отличный инструментарий для создания серверных приложений. Наличие самых разных функций, которые обычно нужно включить в серверное приложение (от интеграции с другими приложениями до сохранения данных в различных базах данных), сильно упрощает жизнь. Неудивительно, что при создании подобных приложений разработчики часто

используют Spring. Этот фреймворк содержит все необходимое для таких реализаций и отлично подходит для любых видов архитектуры. Возможности применения Spring для разработки серверных приложений показаны на рис. 1.6.

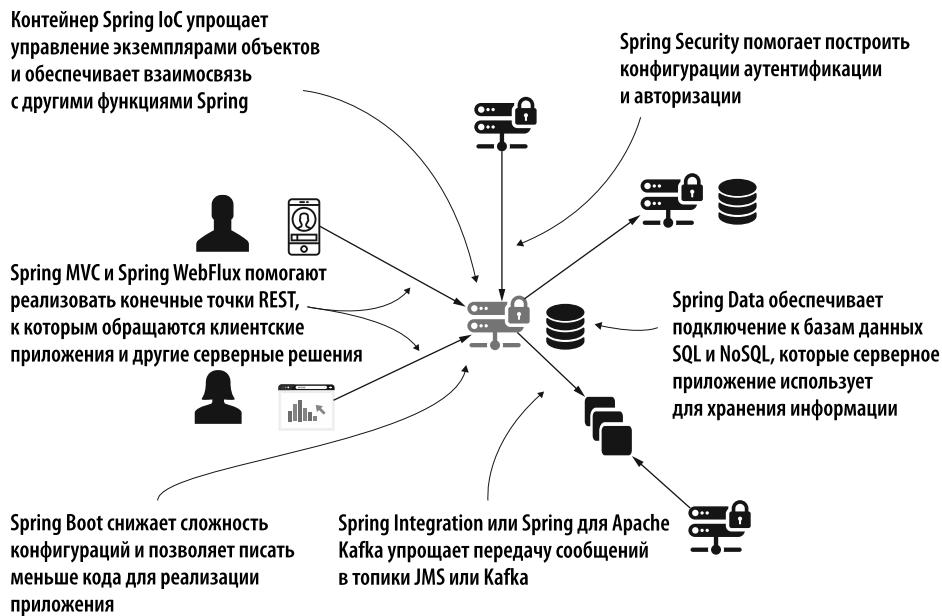


Рис. 1.6. Возможности применения Spring в серверных приложениях практически безграничны: от предоставления функций, которые могут вызываться другими приложениями, до управления доступом к базе данных и от обеспечения безопасности приложения до управления интеграцией посредством сторонних брокеров сообщений

1.3.2. Использование Spring для средств автоматизации тестирования

В наши дни для сквозного тестирования разрабатываемых систем часто используется автоматизированное тестирование. Автоматизация тестирования означает создание программного обеспечения, которое позволяет команде разработчиков убедиться, что поведение приложения соответствует ожидаемому. Команда может составить расписание автоматизированного тестирования таким образом, чтобы проверять приложение достаточно часто и отправлять разработчикам уведомления, если что-то пошло не так. Такая функциональность дает больше уверенности, что создатели вовремя получат сообщение, если при разработке нового функционала они что-то сломают в уже существующих опциях приложения.

В случае небольших систем можно обойтись тестированием вручную, однако выполнение тестовых сценариев всегда полезно автоматизировать. В случае более сложных систем протестировать все потоки вручную в принципе невозможно. Поскольку потоков очень много, выполнение этой задачи заняло бы впечатительное количество времени и слишком много сил.

На практике самым эффективным решением оказывается организация отдельной команды разработчиков, которая создает приложение, отвечающее за валидацию всех потоков тестируемой системы. Когда в систему добавляют новый функционал, тестовое приложение также дополняется с учетом нововведений, и разработчики используют его, чтобы убедиться, что все остальное по-прежнему работает, как должно.

В итоге разработчики используют средство интеграции, с помощью которого составляют расписание регулярного выполнения приложения, чтобы оперативно получать обратную связь о происходящих изменениях (рис. 1.7).

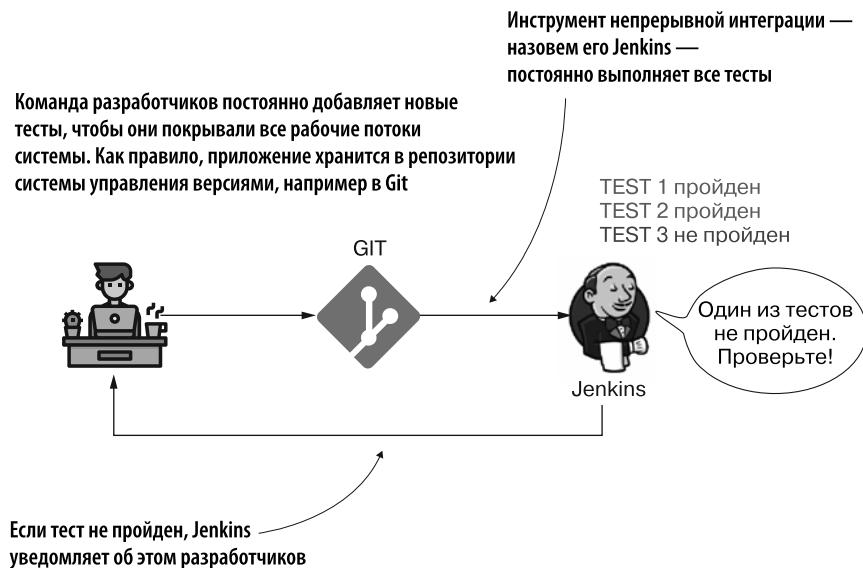


Рис. 1.7. Разработчики развертывают тестовое приложение в тестовой среде. Инструмент непрерывной интеграции наподобие Jenkins постоянно выполняет приложение и отправляет уведомления разработчикам. Таким образом, разработчики всегда в курсе состояния системы и немедленно узнают, если что-нибудь сломается в процессе разработки

Подобные приложения бывают не менее сложны, чем серверные. Чтобы проверить все рабочие потоки, приложение должно взаимодействовать со всеми компонентами системы и даже с базами данных. Иногда оно имитирует внешние зависимости, чтобы создать видимость разных внешних сценариев. Для

написания тестовых сценариев разработчики используют фреймворки Selenium, Cucumber, Gauge и др. Однако даже с ними приложение может получить несколько преимуществ от использования инструментария Spring. Например, с помощью контейнера Spring IoC появляется возможность управлять экземплярами объектов, чтобы получился код, более удобный в обслуживании. При необходимости валидации данных можно организовать подключение к базам данных с помощью Spring Data. Чтобы имитировать определенные сценарии, можно отправлять сообщения в очереди или топики системы брокеров или же использовать Spring просто для вызова конечных точек REST (рис. 1.8). (Напомню еще раз: ничего страшного, если это кажется вам слишком сложным; все прояснится по мере того, как вы будете дальше читать книгу.)

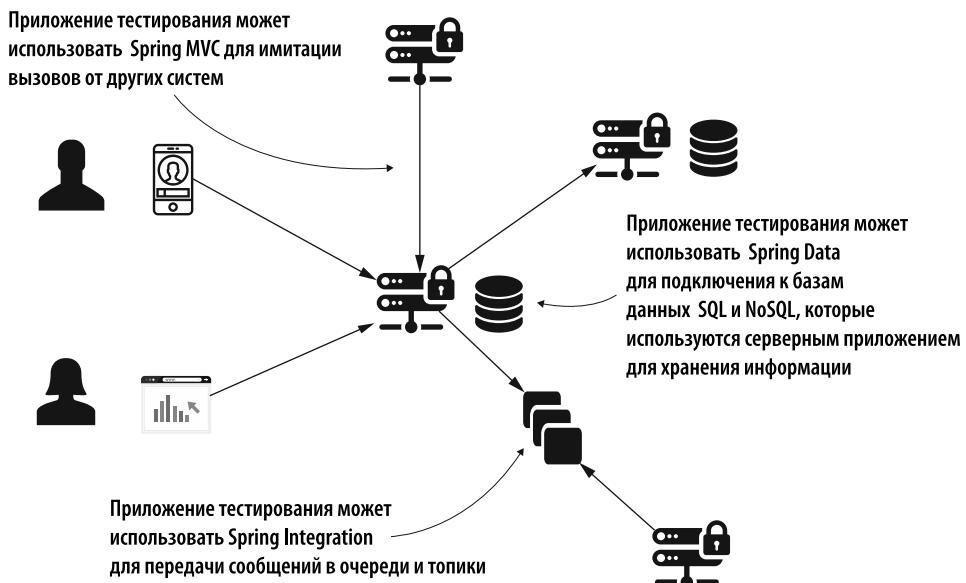


Рис. 1.8. Иногда приложению тестирования нужно подключаться к базам данных, соединяться с другими системами или тестируемой системой. Компоненты экосистемы Spring позволяют упростить реализацию данного функционала

1.3.3. Использование Spring для разработки десктопных приложений

В наши дни десктопные приложения создаются не так уж часто: их обязанности по взаимодействию с пользователем взяли на себя мобильные и веб-приложения. Однако десктопные приложения, хоть их и мало, по-прежнему существуют, и компоненты экосистемы Spring могут оказаться весьма полезными при разработке их функционала. В десктопном приложении можно с успехом использовать контейнер Spring IoC для управления экземплярами объектов. Так реализация

приложения получается более прозрачной, и его удобнее поддерживать. А еще Spring можно использовать для включения различных функций, таких как коммуникация с серверным приложением и другими компонентами (вызов веб-сервисов или использование других технологий для удаленных соединений) или внедрение решений для кеширования.

1.3.4. Использование Spring в мобильных приложениях

Благодаря проекту Spring for Android (<https://Spring.io/projects/Spring-android>) сообщество Spring может помочь в разработке мобильных приложений. Даже если вам редко приходилось с ними сталкиваться, следует помнить о возможности использовать инструментарий Spring при создании продуктов для Android. Этот проект Spring включает в себя REST-клиент для Android и поддержку аутентификации для доступа к защищенным API.

1.4. КОГДА НЕ СТОИТ ИСПОЛЬЗОВАТЬ ФРЕЙМВОРКИ

Рассмотрим вопрос, почему иногда фреймворков следует избегать. Очень важно понимать, когда стоит использовать фреймворк, а когда от него лучше отказаться. Иногда применение инструмента, слишком сложного для конкретной задачи, может потребовать больше сил, чем если бы его не использовали в принципе, или вообще привести к неверному результату. Это все равно что резать хлеб бензопилой: конечно, вы можете попытаться — и у вас даже получится, но процесс выйдет труднее и затратнее, чем нарезка обычным ножом (а вместо нарезанного хлеба может получиться куча крошек). Мы рассмотрим несколько сценариев, в которых использование фреймворка не лучшая идея. А затем я поделюсь историей о том, как одной команде разработчиков, в которую входил и я, не удалось реализовать проект из-за того, что мы использовали фреймворк.

Оказывается, что на фреймворк, как и на все остальное в мире разработки программного обеспечения, нельзя полагаться во всех случаях жизни. В практике вам встретятся ситуации, когда данное решение окажется неподходящим. Или в целом фреймворк будет хорошим выбором, но не фреймворк Spring. В каком из следующих сценариев его не стоит использовать?

1. Требуется реализовать определенный функционал, который бы занимал как можно меньше места (под занимаемым местом я понимаю объем памяти, занятый файлами приложения).
2. Специфические требования по безопасности вынуждают вас писать весь код приложения самостоятельно, без использования каких-либо фреймворков с открытым кодом.
3. Необходимо сделать так много надстроек над фреймворком, что написанного кода получится меньше, если фреймворк не использовать вообще.

4. У вас уже есть работающее приложение; перестроив его на использование фреймворка, вы не получите никаких преимуществ.

Рассмотрим эти пункты подробнее.

1.4.1. Приложение должно занимать как можно меньше места

Здесь я имею в виду ситуацию, когда необходимо маленькое приложение. В современных системах сервисы все чаще доставляются в контейнерах. Вам, вероятно, приходилось слышать о таких контейнерах, как Docker, Kubernetes и подобных (если нет — не страшно).

В целом тема контейнеров выходит за рамки данной книги, поэтому единственное, что вам стоит сейчас знать: если приложение развертывается подобным образом, необходимо, чтобы его размер был как можно меньше. Контейнер по-добен коробке, в которой находится приложение. Важнейший принцип развертывания приложений в контейнерах гласит: контейнеры должны уничтожаться и создаваться максимально быстро. И здесь огромное значение имеет размер приложения. Чем он меньше, тем больше секунд вы сэкономите при инициализации приложения. Впрочем, это не означает, что вообще нельзя использовать фреймворки для приложений, которые развертываются в контейнерах.

Однако для определенных приложений, которые и сами по себе, как правило, довольно малы, предпочтительнее ускорить инициализацию и сократить размер, чем добавлять зависимости от различных фреймворков. Это касается в том числе так называемых *внесерверных функций*. Внесерверные функции представляют собой крошечные приложения, которые развертываются в контейнерах. Как следует из названия, они выполняются вне сервера, и мы едва ли можем влиять на способ их развертывания. Эти приложения обязаны быть маленькими, и именно поэтому, в силу их специфики, при разработке следует по возможности избегать подключения фреймворков. Возможно, что вследствие их размера фреймворк в принципе не понадобится.

1.4.2. Безопасность требует написать весь код самостоятельно

Во втором пункте списка я имел в виду специфические ситуации, когда в приложении нельзя использовать фреймворк вследствие требований по безопасности. Такие требования обычно предъявляются к приложениям, которые создаются для военных или правительственный организаций. Снова отмечу: это вовсе не означает, что воздействие фреймворков запрещено для всех приложений, используемых правительством. Но для некоторых из них существуют ограничения. Вы спросите почему? Хорошо, предположим, что мы применили в подобном приложении

фреймворк с открытым исходным кодом, такой как Spring. Если некто найдет в нем подходящую уязвимость, о которой станет известно, хакеры смогут использовать данную информацию, чтобы взломать приложение. Иногда правообладатели таких продуктов желают свести вероятность взлома их системы к нулю, насколько возможно. Поэтому иногда приходится даже переписывать функционал, вместо того чтобы брать готовый из сторонних источников.

ПРИМЕЧАНИЕ

Но как же так? Ведь раньше я говорил, что использовать открытый фреймворк безопаснее, поскольку если в нем возникает уязвимость, то кто-нибудь обязательно ее заметит. Да, но если инвестировать в проект достаточно много времени и денег, то вы, скорее всего, и сами решите эту проблему. Как правило, фреймворк дешевле. Если нет особых противопоказаний, то лучше использовать фреймворк. Но бывают проекты, правообладатели которых всерьез хотят застраховаться от утечки информации.

1.4.3. Использовать фреймворк нецелесообразно из-за слишком большого количества настроек

Еще один случай, когда, возможно, стоит отказаться от использования фреймворка, — это когда приходится слишком сильно изменять его компоненты и в итоге с ним получается еще больше кода, чем без него. Как я уже отмечал в разделе 1.1, фреймворк предоставляет разработчику функционал, который в сочетании с подключенным к нему бизнес-кодом образует приложение. Эти компоненты, взятые из фреймворка в чистом виде, не всегда подходят идеально, их так или иначе приходится изменять. В целом модифицировать элементы фреймворка и способ их подключения, вместо того чтобы писать все с нуля, — обычная практика. Но если вы обнаружите, что изменений слишком много, — скорее всего, вы неудачно выбрали фреймворк (поиските другие варианты) или, возможно, вам вообще не стоило его использовать.

1.4.4. Когда переход на фреймворк не приносит пользы

В последнем пункте списка я отметил, что попытка использовать фреймворк, чтобы заменить уже существующую рабочую часть приложения, может оказаться ошибочной. Иногда возникает соблазн применить вместо имеющейся архитектуры что-то новое. Появился новый популярный фреймворк, все его используют, так почему бы и нам не переделать наше приложение, чтобы оно тоже опиралось на этот фреймворк? Да, так можно поступить, но прежде необходимо тщательно проанализировать, что вы хотите получить, заменив то, что и так работает. В некоторых случаях, как в моей истории из раздела 1.1, действительно имеет смысл переработать приложение, чтобы оно опиралось на определенный фреймворк. Если это изменение принесет пользу — вперед! Среди возможных причин может быть то, что приложение станет более удобным в обслуживании,

более производительным или более защищенным. Но если изменение не принесет пользы, если, может, оно даже станет причиной нестабильной работы приложения, то в итоге может оказаться, что вы потратили время и деньги на неверный результат. Позвольте рассказать вам историю из моего собственного опыта.

ОШИБКА, КОТОРУЮ МОЖНО БЫЛО ИЗБЕЖАТЬ

Использование фреймворка не всегда удачный вариант, и мне пришлось дорого заплатить за это знание. Много лет назад мы разрабатывали серверную часть веб-приложения. Времени подвластно все, включая архитектуру программного обеспечения. В приложении использовалось прямое подключение к базе данных Oracle посредством JDBC, и код выглядел весьма скверно. Везде, где в приложении требовалось выполнить запрос к базе данных, открывался оператор — и отправленный запрос часто занимал несколько строк. Если вы достаточно молоды, то, возможно, и не застали такую технологию. Тогда просто поверьте: это был длинный и уродливый код. В те времена появилось несколько фреймворков, использующих другой способ работы с базой данных, и они становились все более популярными. Тогда я впервые услышал о Hibernate — ORM-фреймворке, который представлял таблицы базы данных и их взаимосвязи в виде объектов и зависимостей между ними. Это позволяло писать меньше кода и обеспечивало более понятную функциональность. Однако при неправильном использовании фреймворк мог замедлить работу приложения, код становился менее интуитивным и даже могли возникать ошибки.

Приложение, над которым велась работа, нуждалось в изменениях. Мы были уверены, что у нас получилось бы улучшить этот уродливый JDBC-код. Мне казалось, что можно было хотя бы сократить число строк. Такое изменение сделало бы код значительно удобнее в обслуживании. Несколько разработчиков, в том числе и я, предложили использовать инструмент Spring под названием JdbcTemplate (вы познакомитесь с ним в главе 12). Но другие жестко настаивали на Hibernate. Он был весьма популярен, так почему бы им не воспользоваться? (Собственно, он до сих пор является таковым, и в главе 13 мы рассмотрим его интеграцию со Spring.) Я понимал, что адаптация кода под совершенно новую методологию будет непростой задачей. Более того, я не видел в этом смысла. Изменения также были сопряжены с повышенным риском возникновения ошибок.

К нашему счастью, мы начали переход с апробации идеи. После нескольких месяцев усилий и стресса команда решила это прекратить.

Проанализировав имеющиеся варианты, мы в итоге выбрали реализацию с использованием JdbcTemplate. Нам удалось написать более ясный код, удалив множество лишних строк, и не пришлось использовать для этого какой-либо новый фреймворк.

1.5. ЧЕМУ ВЫ НАУЧИТЕСЬ, ПРОЧИТАВ ЭТУ КНИГУ

Раз уж вы открыли эту книгу, то вы, скорее всего, являетесь разработчиком программного обеспечения для экосистем Java, который посчитал полезным изучить Spring. Цель этой книги — научить вас основам работы с данным фреймворком. Предполагается, что вы ничего не знаете о фреймворках в целом и о Spring в частности. Под термином Spring я понимаю не только ядро Spring, но и всю экосистему Spring.

После изучения данной книги вы научитесь делать следующее:

- использовать контекст Spring и создавать аспекты для объектов под управлением фреймворка;
- использовать технологию соединения Spring-приложения с базой данных и работать с сохраненными данными;
- организовывать обмен данными между приложениями с помощью реализованных в Spring интерфейсов REST API;
- строить простейшие приложения на основе концепции «соглашение важнее конфигурации»;
- использовать наилучшие технологии создания типичных классов Spring-приложений;
- тщательно тестировать ваши Spring-разработки.

РЕЗЮМЕ

- Прикладной фреймворк — это набор типичного программного функционала, фундамент для разработки приложения. Фреймворк играет роль скелета, на котором строится приложение.
- Фреймворк помогает создать более эффективный продукт, предоставляя функционал, который вы встраиваете в свою реализацию, а не разрабатываете самостоятельно.
- Использование фреймворка экономит время и уменьшает вероятность возникновения ошибок.
- Использование широко известного фреймворка, в том числе Spring, открывает доступ к помощи огромного сообщества, участники которого, скорее всего, уже сталкивались с теми же проблемами, что и вы. Это отличная возможность узнать, как другие решали знакомые вам вопросы, не проводить собственные исследования и сэкономить время.
- При разработке приложения всегда продумывайте все варианты действий, включая отказ от фреймворка. Решив использовать один или несколько

фреймворков, рассмотрите также все альтернативы. Учтите назначение фреймворка, выясните, кто еще его использует (насколько велико это сообщество) и как давно фреймворк присутствует на рынке (насколько это зрелый проект).

- Spring – это не просто фреймворк. Часто, упоминая Spring в значении «фреймворк Spring», мы подразумеваем его базовый функционал. Однако Spring – это целая экосистема, образованная множеством проектов для разработки приложений. У каждого из них своя область применения; при создании продукта, чтобы реализовать требуемые параметры, можно использовать несколько таких проектов. В этой книге нам понадобятся следующие:
 - Spring Core – основа Spring, предоставляющая такие функции, как контекст, аспекты и базовый доступ к данным;
 - Spring Data – набор качественного удобного инструментария для реализации уровня доступа к данным в приложении. Вы оцените, как легко использовать Spring Data для подключения к базам данных SQL и NoSQL;
 - Spring Boot – проект, помогающий реализовать концепцию «соглашения важнее конфигурации».
- В учебных материалах (книгах, статьях, видеоучебниках) использование Spring обычно ограничивается серверными приложениями. Данный фреймворк действительно получил широкое распространение в их разработке, однако он также будет полезен при создании продуктов других типов, даже десктопных приложений и средств автоматизации тестирования.

Контекст Spring: что такое бины

В этой главе

- ✓ Зачем нужен контекст Spring.
- ✓ Как добавлять в контекст Spring новые объекты.

В этой главе мы начнем изучать главный элемент фреймворка Spring — контекст (в Spring-приложениях его еще называют контекстом приложения). Представьте контекст как место в памяти приложения, куда добавляются все экземпляры объектов, которыми должен управлять фреймворк. По умолчанию Spring ничего о них не знает. Чтобы фреймворк эти объекты «увидел», их нужно добавить в контекст. Далее в книге мы рассмотрим, как применить различные возможности, которые появляются у продукта благодаря Spring. Вы узнаете, что все они становятся доступными через контекст — для этого в контекст добавляются экземпляры объектов и устанавливаются связи между ними. Spring использует их для подключения к приложению собственных функций. В данном издании вы познакомитесь с основными вариантами применения ключевых возможностей Spring, таких как транзакции, тестирование и т. п.

Ответив на вопросы, что представляет собой контекст Spring и как он работает, вы сделаете первый шаг в использовании данного фреймворка. Не умея управлять контекстом Spring, невозможно сделать практически ничего из того, о чем вы узнаете позже. Контекст — сложный механизм, позволяющий Spring контролировать создаваемые вами экземпляры. Таким образом, контекст позволяет использовать возможности фреймворка.

В настоящей главе мы начнем с того, что научимся добавлять экземпляры объектов в контекст Spring. В главе 3 вы узнаете, как ссылаться на добавленные в контекст экземпляры и устанавливать связи между ними.

Эти экземпляры объектов мы будем называть бинами (beans). И конечно же, чтобы освоить синтаксические конструкции, мы будем писать фрагменты кода. Вы найдете их в проектах, которые предоставляются в комплекте с данным изданием (эти проекты можно загрузить по адресу <https://manning-content.s3.amazonaws.com/download/a/32357a2-2420-4c0f-be67-645246ae0d94/code.zip>). Я буду сопровождать примеры кода иллюстрациями и подробными пояснениями использованных приемов.

Поскольку я хочу, чтобы ваше знакомство со Spring было последовательным, в этой главе мы остановимся на тех синтаксических конструкциях, которые необходимо знать для работы с контекстом Spring. Позже вы обнаружите, что не все объекты приложения должны находиться под контролем фреймворка, так что нет необходимости добавлять в контекст каждый из них. Но пока предлагаю вам сконцентрироваться на изучении разных способов добавления экземпляра в Spring, чтобы затем можно было этим экземпляром управлять.

2.1. СОЗДАНИЕ ПРОЕКТА MAVEN

Рассмотрим создание проекта Maven. Maven не имеет прямого отношения к Spring, однако с помощью этого инструмента легко управлять процессом сборки приложения независимо от используемого фреймворка. Простейшие навыки работы с Maven понадобятся вам для того, чтобы выполнять примеры из настоящей книги. Maven также является одним из самых популярных инструментов разработки Spring-проектов (второе место занимает сборщик Gradle, но мы не будем его рассматривать). Возможно, вы уже знакомы с данной системой и знаете, как создать в ней проект и добавить зависимости, используя файл конфигурации. В таком случае можете пропустить этот раздел и сразу перейти к разделу 2.2.

Система сборки — это программное обеспечение, которое упрощает сборку приложений. Систему сборки можно настроить на автоматическое выполнение различных задач, являющихся частью процесса сборки, вместо того чтобы разбираться с ними вручную. Вот несколько примеров задач, которые часто выполняются при сборке приложения:

- загрузка зависимостей, необходимых для работы приложения;
- выполнение тестов;
- проверка соответствия синтаксиса заданным правилам;
- проверка на наличие уязвимостей в системе безопасности;
- компиляция приложения;
- упаковка приложения в исполняемый архив.

В наших проектах мы будем использовать систему сборки, чтобы было проще управлять зависимостями. В этом разделе будет показано только то, что необходимо для выполнения заданий данной книги: мы поэтапно пройдем через процесс создания проекта в системе Maven, и я расскажу вам самое важное о его структуре. Если хотите изучить Maven более подробно, советую прочитать издание *Introducing Maven: A Build Tool for Today's Java Developers* by Balaji Varanasi (APress, 2019).

Итак, начнем с самого начала. Прежде всего, как при создании любого приложения, нам понадобится интегрированная среда разработки (integrated development environment, IDE). Все современные профессиональные IDE поддерживают проекты Maven, так что можете выбирать любую: IntelliJ IDEA, Eclipse, Spring STS, Netbeans и т. д. В книге я использовал IntelliJ IDEA, так как именно эту среду я применяю чаще всего. Не беспокойтесь — структура проекта Maven всегда одна и та же, независимо от выбранной IDE.

Для начала создадим новый проект. В IntelliJ проект создается с помощью команды **File > New > Project**. После этого у вас должно появиться окно, подобное изображенному на рис. 2.1.

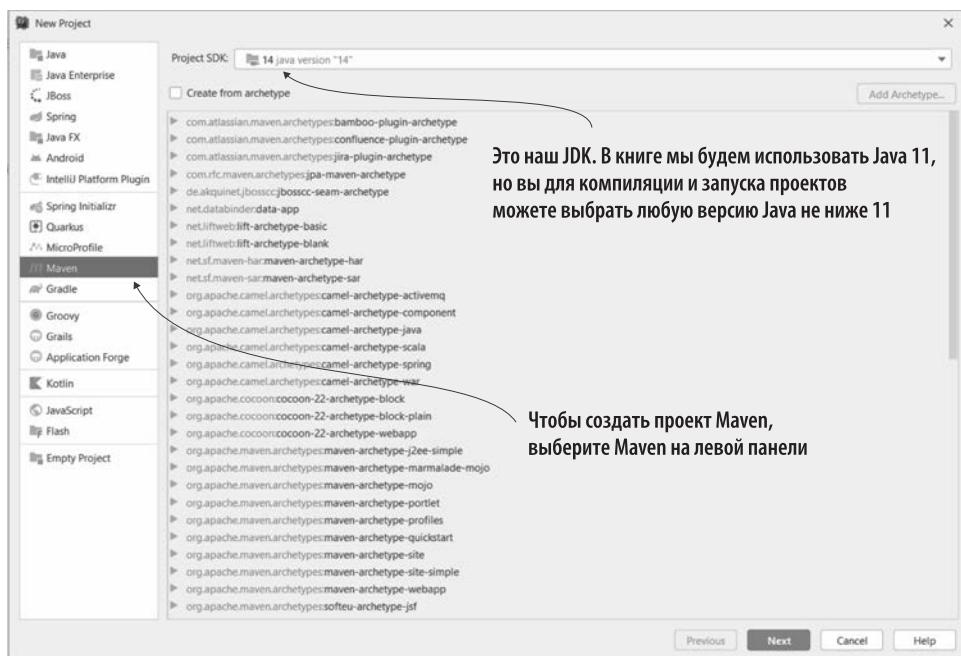


Рис. 2.1. Создание проекта Maven. После того как вы выберете команду **File > New > Project**, откроется вот такое окно, в котором на левой панели следует выбрать тип проекта. В данном случае выбираем Maven. Вверху окна нужно выбрать JDK, который будет использоваться для компиляции и запуска проекта

После того как вы выберете тип проекта, в следующем окне (рис. 2.2) нужно вписать имя проекта. Кроме имени и места, где он будет храниться, для проекта Maven также можно указать следующее:

- ID группы, который используется для объединения нескольких взаимосвязанных проектов;
- ID продукта — имя текущего приложения;
- версию — идентификатор текущей реализации.

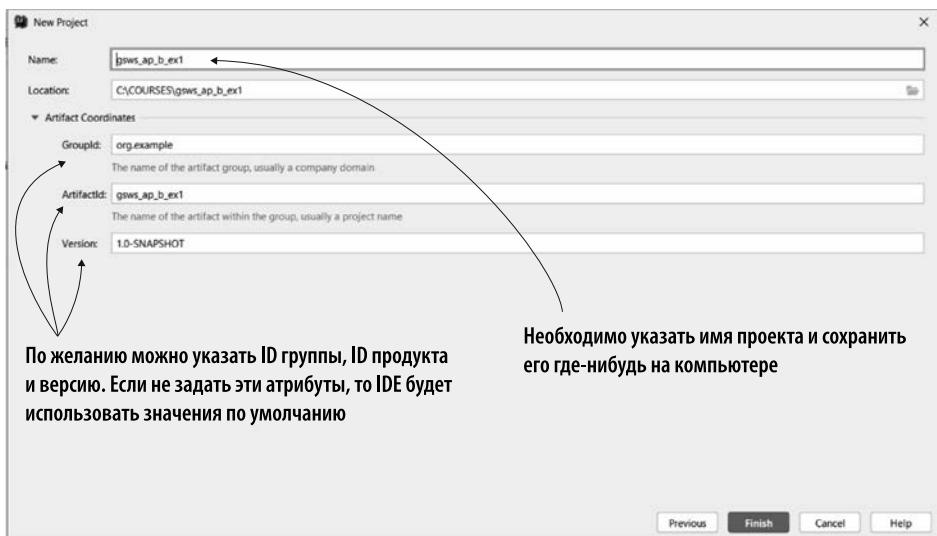


Рис. 2.2. Прежде чем завершить создание проекта, необходимо указать его имя и место хранения в IDE. При желании также можно присвоить проекту ID группы, ID продукта и версию. После этого нажмите кнопку Finish в нижнем правом углу, чтобы закончить создание проекта

В реальных приложениях эти три атрибута очень важны, и их необходимо указывать. Но в нашем случае, поскольку это всего лишь теоретические примеры, их можно пропустить, позволив IDE заполнить поля значениями по умолчанию.

Когда проект будет создан, вы увидите структуру, подобную той, что показана на рис. 2.3. Напомню: структура проекта Maven не зависит от IDE, выбранной для работы. Когда вы впервые увидите свой проект, то заметите два основных момента:

- папку `src` (от source folder — «папка исходного кода»), в которой размещается все, что имеет отношение к проекту;
- файл `pom.xml`, в который записываются параметры конфигурации проекта Maven, — именно сюда мы будем добавлять новые зависимости.

Внутри папки `src` находятся следующие папки:

- папка `main`, в которой хранится исходный код приложения. В ней содержатся код Java и параметры конфигурации — в подпапках `Java` и `resources` соответственно;
- папка `test`, в которой хранится исходный код модульных тестов (подробнее о модульных тестах и о том, как их создавать, мы поговорим в главе 15).

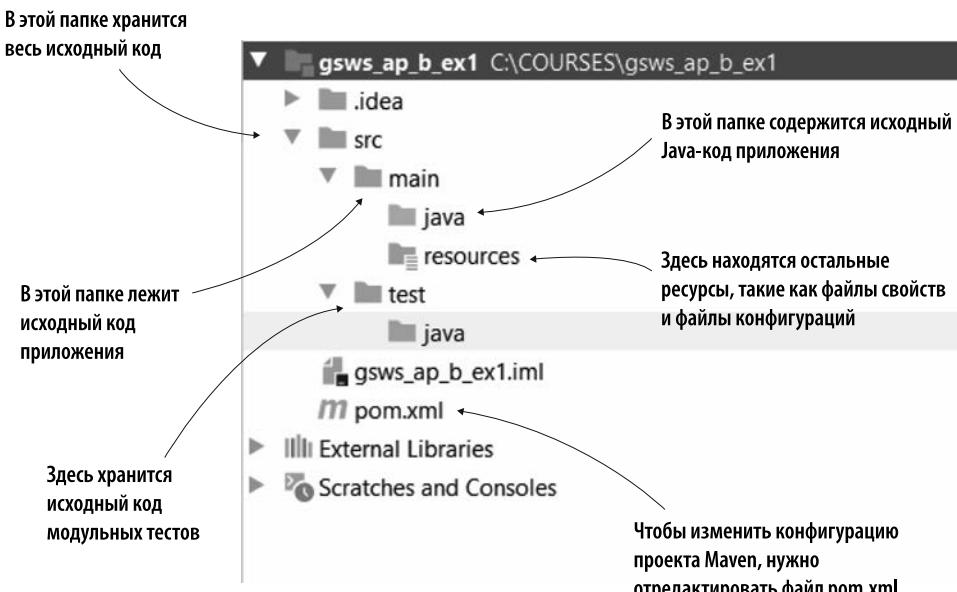


Рис. 2.3. Структура проекта Maven. В папке `src` хранится все, что относится к приложению: исходный код приложения — в папке `main`, исходный код модульных тестов — в папке `test`. В `pom.xml` записывается конфигурация проекта Maven (в наших примерах мы будем использовать этот файл главным образом для определения зависимостей)

На рис. 2.4 показано, как добавить исходный код в папку `main/Java` проекта Maven. В папке появляются новые классы приложения.

В проектах, которые мы будем создавать в рамках этой книги, мы будем использовать множество внешних зависимостей — библиотеки и фреймворки для реализации функционала рассматриваемых примеров. Чтобы добавить эти зависимости в проект Maven, необходимо изменить файл `pom.xml`. В листинге 2.1 показано, как выглядит `pom.xml` сразу после создания проекта Maven.

В папке Java находятся обычные пакеты и классы Java, которые создаются для проекта. В данном случае я создал пакет main и новый класс Main внутри него

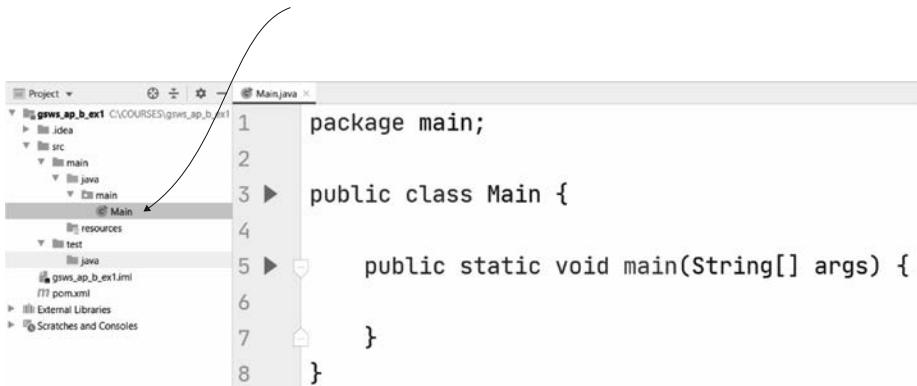


Рис. 2.4. В папке Java размещаются обычные пакеты и классы Java для вашего приложения. Именно эти классы определяют логику приложения и используют создаваемые вами зависимости

Листинг 2.1. Содержимое файла pom.xml по умолчанию

```

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

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>sq-ch2-ex1</artifactId>
    <version>1.0-SNAPSHOT</version>
</project>

```

Проект с таким pom.xml не содержит ни одной внешней зависимости. Заглянув в папку внешних зависимостей проекта, мы увидим там только JDK (рис. 2.5).

В листинге 2.2 показано, как добавить в проект внешнюю зависимость. Все зависимости записываются между тегами <dependencies> и </dependencies>. Каждая зависимость представлена группой тегов <dependency> и </dependency>, внутри которых указываются атрибуты зависимости: ID группы, ID продукта и номер версии. Maven ищет зависимости по этим трем параметрам и загружает найденные зависимости из репозитория. Я не буду углубляться в детали настройки индивидуального репозитория. Просто учите, что по умолчанию Maven

берет зависимости (обычно это JAR-файлы) из репозитория с названием Maven central. Загруженные JAR-файлы размещаются в папке внешних зависимостей проекта, как показано на рис. 2.6.

Пока в разделе External Libraries нового проекта есть только JDK. По мере добавления в проект зависимостей здесь будут появляться новые файлы, соответствующие внешним зависимостям

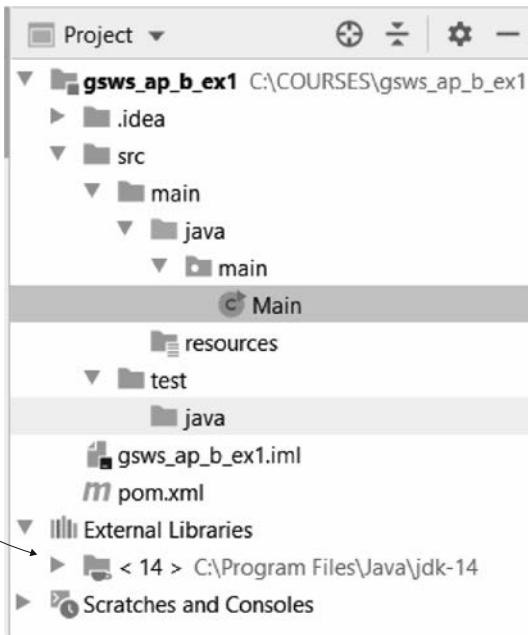


Рис. 2.5. По умолчанию, согласно данным из файла pom.xml, единственной внешней зависимостью проекта является JDK. Одна из причин, по которой приходится редактировать pom.xml (и именно поэтому мы будем его исправлять в данной книге), — добавление новых зависимостей, необходимых для работы приложения

Листинг 2.2. Добавление зависимости в файл pom.xml

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

    <groupId>org.example</groupId>
    <artifactId>sq_ch2_ex1</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies> ←
        <dependency> ←
            <groupId>org.springframework</groupId>
            <artifactId>spring-jdbc</artifactId>

```

Все зависимости проекта
записываются между тегами
<dependencies> и </dependencies>

Описание зависимости
представляет собой группу тегов
<dependency> ... </dependency>

```
<version>5.2.6.RELEASE</version>
  </dependency>
</dependencies>
</project>
```

Если добавить зависимость в файл pom.xml, как в предыдущем листинге, то IDE ее загрузит и зависимость появится в папке External Libraries (рис. 2.6).

При добавлении зависимости для контекста Spring в папке внешних зависимостей проекта появляется несколько новых файлов

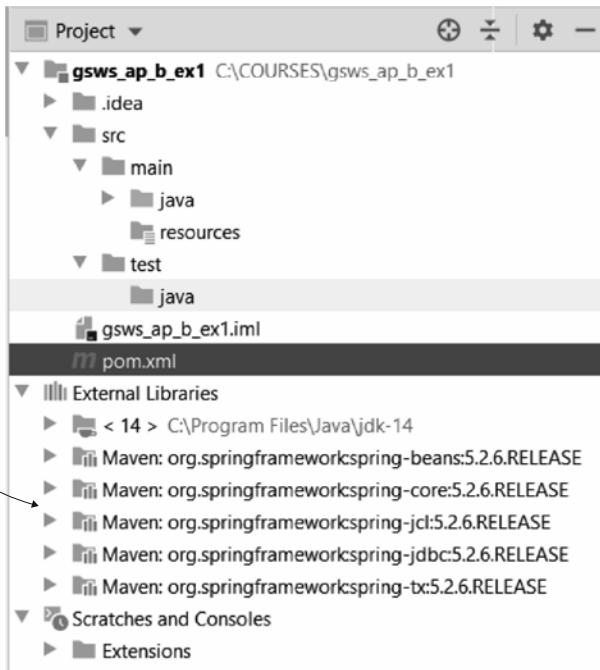
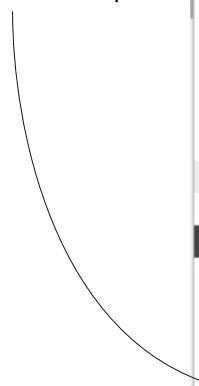


Рис. 2.6. При добавлении зависимости в pom.xml Maven загружает jar-файлы, описывающие эту зависимость. jar-файлы появляются в папке проекта External Libraries

Теперь можно переходить к следующему пункту, где мы рассмотрим основы работы с контекстом Spring. Вы будете создавать проекты Maven и научитесь использовать зависимость Spring, называемую `spring-context`, для управления контекстом.

2.2. ДОБАВЛЕНИЕ БИНОВ В КОНТЕКСТ SPRING

Далее вы научитесь добавлять экземпляры объектов (так называемые бины) в контекст Spring. Вы поймете, что для этого существует несколько способов. В результате Spring сможет управлять бинами и подключать к приложению предоставляемые ими функции. Вы научитесь выбирать способ добавления бинов, адекватный конкретной ситуации: мы определимся, когда лучше взять

тот или иной. Существуют следующие способы включить бин в контекст (далее мы рассмотрим их подробнее):

- посредством аннотации `@Bean`;
- посредством стереотипных аннотаций;
- программно.

Вначале мы создадим проект вообще без ссылок на Spring или какой-либо другой фреймворк. А затем добавим в него зависимости, необходимые, чтобы начать использовать контекст Spring — и создадим этот контекст (рис. 2.7). Данный пример будет хорошей подготовкой к следующим задачам (рассматриваемым в подразделах 2.2.1–2.2.3), где будем добавлять бины в контекст Spring.

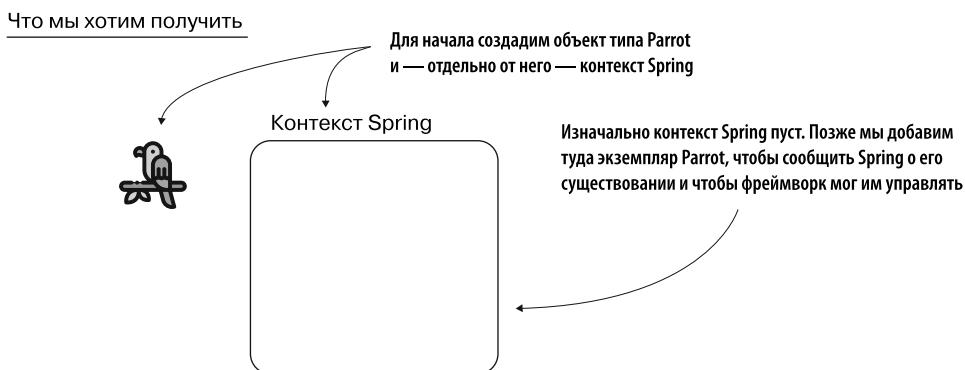


Рис. 2.7. Для начала создадим экземпляр объекта и пустой контекст Spring

Мы создадим проект Maven и определим в нем класс. Чтобы было веселее, предлагаю использовать класс `Parrot` (попугай) с единственным атрибутом типа `String`, представляющим собой кличку попугая (листинг 2.3). Напомню: в этой главе мы будем только добавлять бины в контекст Spring, поэтому будет полезен любой объект, который поможет быстрее запомнить синтаксические конструкции. Код этого примера находится в проекте `sq-ch2-ex1`. В своем проекте вы можете использовать то же имя или выбрать другое, на свой вкус.

Листинг 2.3. Класс Parrot

```
public class Parrot {
    private String name;
    // здесь должны быть геттеры и сеттеры
}
```

Теперь можно определить класс, который будет содержать метод `main`, и создать экземпляр класса `Parrot`, как показано в листинге 2.4. Я обычно называю такой класс `main`.

Листинг 2.4. Создание экземпляра класса Parrot

```
public class main {

    public static void main(String[] args) {
        Parrot p = new Parrot();
    }
}
```

Теперь пора добавить в наш проект необходимые зависимости. Поскольку мы используем Maven, то будем добавлять зависимости в файл `pom.xml`, как показано в листинге 2.5.

Листинг 2.5. Добавление зависимостей для контекста Spring

```
<project xmlns="http://Maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://Maven.apache.org/POM/4.0.0
                             http://Maven.apache.org/xsd/Maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>sq-ch2-ex1</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>org.Springframework</groupId>
            <artifactId>Spring-context</artifactId>
            <version>5.2.6.RELEASE</version>
        </dependency>
    </dependencies>
</project>
```

Главное, на что здесь нужно обратить внимание, — это то, что Spring является модульной конструкцией. Под модульной конструкцией я подразумеваю следующее: если вы используете что-то из экосистемы Spring, то не обязаны подключать к приложению весь Spring целиком, достаточно только нужных вам частей. Поэтому, как видно из листинга 2.5, я добавил лишь зависимость `spring-context`, при наличии которой Maven загрузит зависимости, необходимые для использования контекста Spring. В данной книге мы будем добавлять в проекты разнообразные зависимости в соответствии с поставленной задачей, но они всегда будут только самые необходимые.

ПРИМЕЧАНИЕ

Возможно, вам интересно, как я понял, какую именно зависимость Maven следует использовать. Дело в том, что мне приходилось ее подключать так часто, что я уже ее запомнил. Однако нет необходимости заучивать все на память. Зависимости, которые нужно добавить вручную при создании проекта Spring, вы найдете в справочнике (<https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/core.html>). Большинство зависимостей Spring относятся к группе с ID org.springframework.

После того как зависимость будет добавлена в проект, можно создать экземпляр контекста Spring. В листинге 2.6 показано, что нужно изменить в методе `main`, чтобы создать экземпляр контекста Spring.

Листинг 2.6. Создание экземпляра контекста Spring

```
public class main {
    public static void main(String[] args) {
        var context =
            new AnnotationConfigApplicationContext(); ←
        Parrot p = new Parrot();
    }
}
```

Создание экземпляра
контекста Spring

ПРИМЕЧАНИЕ

Для создания экземпляра контекста Spring мы использовали класс `AnnotationConfigApplicationContext`. У Spring есть несколько реализаций. Поскольку в большинстве случаев вы будете применять класс `AnnotationConfigApplicationContext` (где используется наиболее распространенный в настоящее время инструмент — аннотации), в своей книге я тоже остановлюсь на нем. Как и всегда, я расскажу только то, что вам действительно необходимо знать. Если вы только начинаете работать со Spring, советую пока что не углубляться в детали различных реализаций контекста и цепочки наследований этих классов. Иначе есть вероятность погрязнуть в несущественных мелочах, вместо того чтобы сконцентрироваться на главном.

Как показано на рис. 2.8, мы создали экземпляр `Parrot`, добавили в проект зависимости контекста Spring и создали экземпляр контекста Spring. Наша цель — добавить в контекст объект `Parrot`, что мы и сделаем на следующем шаге.

На этом мы закончили формировать заготовку (скелет) проекта, которую будем использовать в следующих пунктах, чтобы научиться добавлять бины в контекст Spring. В подразделе 2.2.1 мы научимся добавлять экземпляр в контекст Spring с помощью аннотации `@Bean`. Далее, в подразделах 2.2.2 и 2.2.3, вы узнаете о других способах добавить экземпляр в контекст — посредством стереотипных аннотаций и программно. Рассмотрев все три способа, мы сравним их, и вы узнаете, какой из них в каких случаях лучше применять.

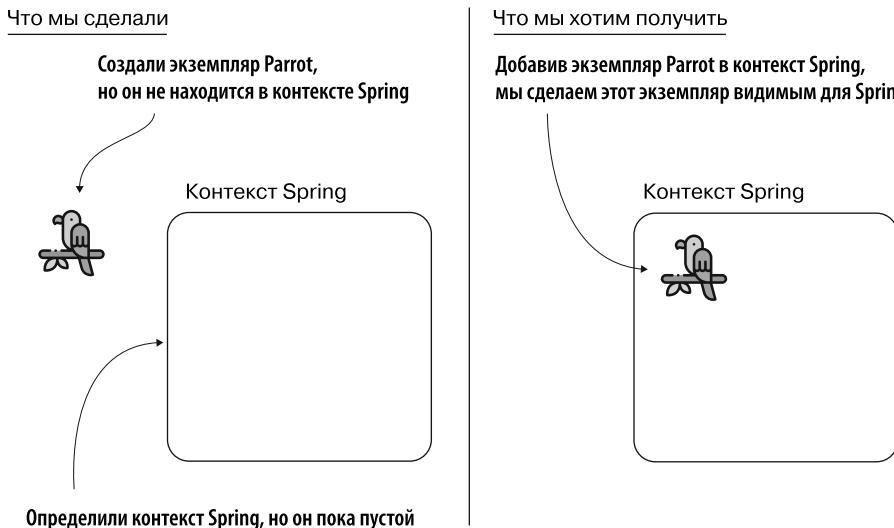


Рис. 2.8. Мы создали экземпляр контекста Spring и экземпляр Parrot. Теперь мы хотим добавить экземпляр Parrot в контекст Spring, чтобы сделать Parrot видимым для Spring

2.2.1. Добавление бинов в контекст Spring с помощью аннотации @Bean

Рассмотрим, как добавить экземпляр объекта в контекст Spring с помощью аннотации `@Bean`. Данный способ позволяет добавлять экземпляры классов, определенных вами в проекте (таких как наш `Parrot`), а также классов, которые вы не создавали сами, но используете в приложении. Считаю, что этот метод самый простой для понимания новичками. Напомню: мы учимся добавлять бины в контекст Spring потому, что Spring может управлять только теми объектами, которые являются частью фреймворка. Вначале я приведу простой пример добавления бина в контекст Spring посредством аннотации `@Bean`, а затем покажу, как добавить несколько бинов одного или разных типов. Чтобы добавить бин в контекст Spring посредством аннотации `@Bean`, нужно сделать следующее (рис. 2.9).

1. Определить в проекте класс конфигурации (с аннотацией `@Configuration`). Этот класс (который мы рассмотрим позже) используется для описания конфигурации контекста Spring.
2. Добавить в класс конфигурации метод, возвращающий экземпляр объекта, который мы хотим добавить в контекст, и снабдить этот метод аннотацией `@Bean`.
3. Настроить Spring на использование класса конфигурации, созданного в пункте 1. Как вы узнаете ниже, классы конфигураций используются для создания различных конфигураций фреймворка.

Выполним эти операции для проекта sq-c2-ex2. Чтобы сохранить каждую операцию отдельно, советую создавать новый проект для каждого примера.

ПРИМЕЧАНИЕ

Напомню, что все проекты из этой книги находятся по адресу <https://manning-content.s3.amazonaws.com/download/a/32357a2-2420-4c0f-be67-645246ae0d94/code.zip>.

ПРИМЕЧАНИЕ

Класс конфигурации – это специальный класс в приложении Spring, посредством которого можно настроить фреймворк на выполнение определенных действий, таких как создание бинов или активация определенного функционала. В данной книге вы узнаете, как много всего можно описать в файле конфигурации.

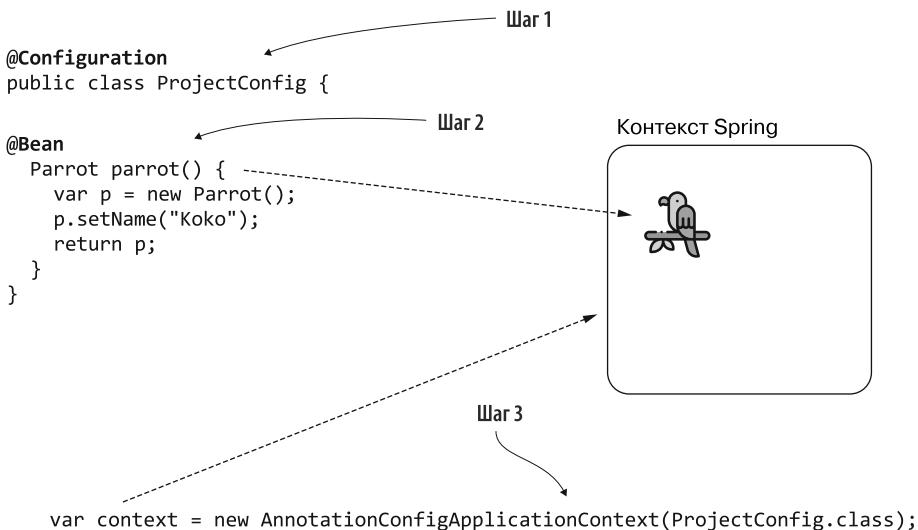


Рис. 2.9. Операции, которые нужно выполнить, чтобы добавить бин в контекст с помощью аннотации `@Bean`. Добавив экземпляр в контекст Spring, мы сообщаем фреймворку об этом объекте, после чего фреймворк сможет им управлять

Шаг 1. Создание файла конфигурации

Прежде всего мы создадим в проекте класс конфигурации. Характерный признак класса конфигурации Spring – аннотация `@Configuration`. С его помощью мы будем определять различные параметры проекта, связанные со Spring. Далее в этой книге вы узнаете, как много всего можно описать в классах конфигурации, но сейчас мы займемся только добавлением новых экземпляров в контекст

Spring. В листинге 2.7 показано, как создать класс конфигурации, которому я присвоил имя `ProjectConfig`.

Листинг 2.7. Определение конфигурационного класса в проекте

```
@Configuration
public class ProjectConfig {
```

Для определения конфигурационного класса Spring используется аннотация `@Configuration`

ПРИМЕЧАНИЕ

Чтобы код был более понятным, я создал для каждого класса отдельный проект — например, конфигурационные классы описал в проекте `config`, а класс `Main` — в проекте `main`. Подобный ход является в разработке хорошим тоном — советую в повседневной практике поступать так же.

Шаг 2. Создание метода, который возвращает бин, с аннотацией `@Bean`

Класс конфигурации позволяет, в частности, добавлять бины в контекст Spring. Для этого нужно определить метод, возвращающий экземпляр объекта, который мы хотим добавить в контекст, и снабдить этот метод аннотацией `@Bean`. Она сообщит Spring о том, что при инициализации контекста нужно вызвать данный метод и добавить возвращенное им значение в контекст. В листинге 2.8 показано, как нужно изменить класс конфигурации, чтобы это реализовать.

ПРИМЕЧАНИЕ

Для проектов, рассматриваемых в книге, я использовал Java 11 — последнюю версию с долговременной поддержкой. Сейчас все больше проектов переходит именно на нее. В целом единственное специфическое свойство, которое я использовал в примерах и которого нет в более ранних версиях Java, — это зарезервированное имя типа `var`. Я постоянно включаю `var`, чтобы получить более короткий и понятный код. Но если вы предпочитаете более раннюю версию Java (например, Java 8), то можете заменить `var` принятым там типом, тогда ваши проекты будут работать и в Java 8.

Листинг 2.8. Определение метода `@Bean`

```
@Configuration
public class ProjectConfig {
    @Bean
    Parrot parrot() {
        var p = new Parrot();
        p.setName("Koko");
        return p;
    }
}
```

Добавив аннотацию `@Bean`, мы сообщаем Spring, что при инициализации контекста нужно вызвать этот метод и добавить в контекст возвращенное им значение

Назначаем имя попугая, которое далее будем использовать при тестировании приложения

Spring добавляет в контекст экземпляр класса `Parrot`, возвращаемый методом

Обратите внимание на то, что имя, которое я присвоил методу, — это не глагол. Как вы, вероятно, знаете, хорошим тоном программирования на Java является использование имен в форме глагола, так как методы обычно описывают некие действия. Однако для методов, добавляющих бины в контекст Spring, мы сделаем исключение. Эти методы описывают экземпляры возвращаемых ими объектов, которые затем становятся частью контекста Spring. Имя метода становится именем бина (например, в листинге 2.8 бин получает имя `parrot`). По соглашению, здесь можно использовать имена существительные, которые обычно совпадают с именем класса.

Шаг 3. Настройка Spring на инициализацию контекста с использованием нового класса конфигурации

Итак, мы создали класс конфигурации, где сообщили Spring о существовании экземпляра объекта, который должен стать бином. Теперь нужно сделать так, чтобы Spring использовал этот класс конфигурации при инициализации контекста. В листинге 2.9 показано, как изменить инициализацию контекста Spring в классе `Main`, чтобы подключить созданный нами ранее класс конфигурации.

Листинг 2.9. Инициализация контекста Spring на основании созданного класса конфигурации

```
public class main {
    public static void main(String[] args) {
        var context =
            new AnnotationConfigApplicationContext(
                ProjectConfig.class); ← При создании экземпляра контекста Spring нужно передать класс конфигурации как параметр — и тогда Spring будет его использовать
    }
}
```

Чтобы убедиться, что экземпляр `Parrot` теперь действительно является частью контекста, можно обратиться к этому экземпляру и вывести его имя в консоль, как показано в листинге 2.10.

Листинг 2.10. Обращение к экземпляру Parrot из контекста

```
public class main {
    public static void main(String[] args) {
        var context =
            new AnnotationConfigApplicationContext(
                ProjectConfig.class);
        Parrot p = context.getBean(Parrot.class); ← Получаем ссылку на бин типа Parrot из контекста Spring
        System.out.println(p.getName());
    }
}
```

Теперь в консоль будет выведено имя, которое мы присвоили попугаю при добавлении в контекст. Я назвал его Koko.

ПРИМЕЧАНИЕ

На практике, чтобы проверить, правильно ли работает приложение, применяются модульные и интеграционные тесты. В этой книге проекты тоже сопровождаются модульными тестами, позволяющими убедиться, что приложения работают именно так, как описано. Поскольку данное издание предназначено для начинающих, вы, возможно, еще не имеете представления о модульных тестах. Чтобы не запутаться и сконцентрироваться на изучаемом материале, мы не будем обсуждать модульные тесты до главы 15. Но если вы уже умеете писать модульные тесты и их чтение поможет вам лучше разобраться в теме, то все они есть в папке `test`, которая находится во всех наших проектах Maven. Если же вы еще не знаете, как работают модульные тесты, советую уделить внимание только обсуждаемому предмету.

Как и в предыдущем примере, вы можете добавить в контекст Spring объект любого типа (рис. 2.10). Добавим туда объекты типа `String` и `Integer` и посмотрим, как это работает.

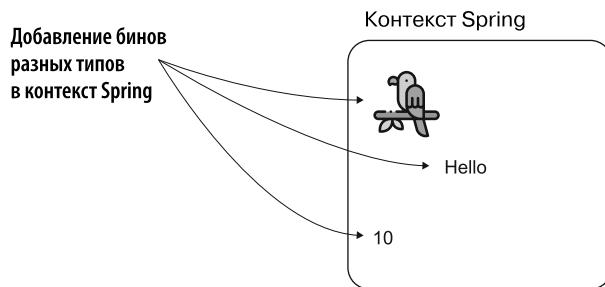


Рис. 2.10. В контекст Spring можно добавить любой объект, чтобы сообщить Spring о существовании этого объекта

В листинге 2.11 показано, какие изменения нужно внести в класс конфигурации, чтобы добавить в контекст бины типа `String` и `Integer`.

Листинг 2.11. Добавление в контекст еще двух бинов

```
@Configuration
public class ProjectConfig {

    @Bean
    Parrot parrot() {
        var p = new Parrot();
        p.setName("Koko");
        return p;
    }
}
```

```

@Bean ←
String hello() {
    return "Hello";
}

@Bean ←
Integer ten() {
    return 10;
}

```

Добавление в контекст Spring
строки "Hello"

Добавление в контекст Spring
целого числа 10

ПРИМЕЧАНИЕ

Напомню о назначении контекста Spring: мы добавляем туда те экземпляры, которыми Spring должен управлять. (Таким образом подключая к ним функционал фреймворка.) В реальных приложениях далеко не все объекты включаются в контекст Spring. Начиная с главы 4, где наши примеры станут более похожими на код реального приложения, мы также уделим больше внимания вопросу, какие объекты нуждаются в управлении Spring. Но пока что сконцентрируемся на методах добавления бинов в контекст Spring.

Теперь мы можем ссылаться на эти два бина так же, как и на экземпляр `parrot`. В листинге 2.12 показано, что нужно изменить в методе `main`, чтобы вывести в консоль значения новых бинов.

Листинг 2.12. Вывод в консоль значений двух новых бинов

```

public class main {

    public static void main(String[] args) {
        var context = new AnnotationConfigApplicationContext(
            ProjectConfig.class);

        Parrot p = context.getBean(Parrot.class); ←
        System.out.println(p.getName());

        String s = context.getBean(String.class);
        System.out.println(s);

        Integer n = context.getBean(Integer.class);
        System.out.println(n);
    }
}

```

Явное преобразование типов
не требуется. Spring ищет
в контексте нужный бин.
Если такого бина не существует,
то Spring выбрасывает
исключение

Теперь, как показано в следующем фрагменте кода, при запуске приложения в консоль выводятся значения трех бинов:

Koko
Hello
10

Итак, мы научились добавлять в контекст Spring один или несколько бинов разных типов. Но можно ли включить туда несколько объектов одного типа (рис. 2.11)? И если можно, то как различать эти объекты? Чтобы продемонстрировать, как добавить в контекст Spring несколько бинов одного типа и как потом на них ссылаться, мы создадим новый проект с именем sq-ch2-ex3.

ПРИМЕЧАНИЕ

Не путайте имя бина и кличку попугая. В нашем примере именами (идентификаторами) бинов в контексте Spring являются `parrot1`, `parrot2` и `parrot3` (имена методов с аннотацией `@Bean`, определяющих соответствующие бины). Я дал попугаям клички `Koko`, `Miki` и `Riki`. Кличка попугая — это просто атрибут объекта `Parrot`, и для Spring она не имеет никакого значения.

Можно объявить любое количество экземпляров одного типа, просто декларируя для каждого свой метод с аннотацией `@Bean`. В листинге 2.13 я объявил в классе конфигурации три бина типа `Parrot`. Этот пример находится в проекте `sq-ch2-ex3`.

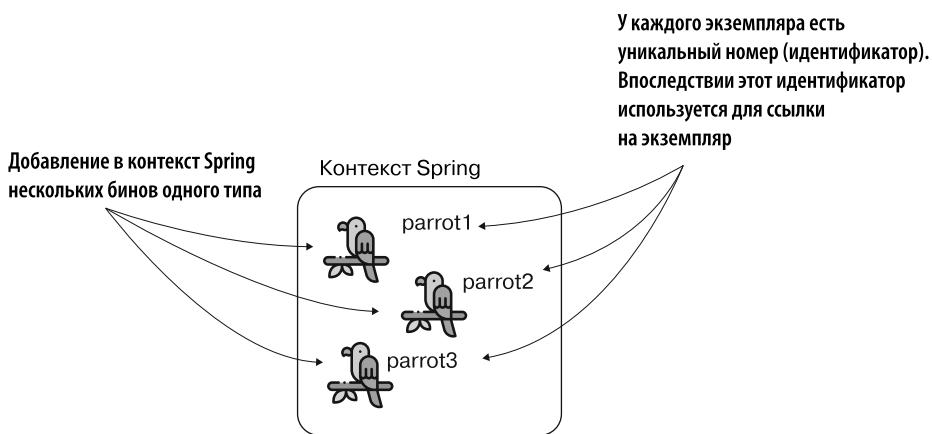


Рис. 2.11. Для добавления в контекст Spring нескольких бинов одного типа используется несколько методов с аннотацией `@Bean`. Каждому экземпляру присваивается уникальный идентификатор. Чтобы впоследствии сослаться на этот экземпляр, нужно использовать идентификатор бина

Листинг 2.13. Добавление в контекст Spring нескольких бинов одного типа

```
@Configuration
public class ProjectConfig {
```

```
    @Bean
    Parrot parrot1 {
```

```

        var p = new Parrot();
        p.setName("Koko");
        return p;
    }

@Bean
Parrot parrot2() {
    var p = new Parrot();
    p.setName("Miki");
    return p;
}

@Bean
Parrot parrot3() {
    var p = new Parrot();
    p.setName("Riki");
    return p;
}
}

```

Понятно, что теперь мы не сможем получать бины из контекста, указав один лишь их тип. Если попробовать так сделать, то получим исключение, поскольку Spring не сможет угадать, на какой именно из объявленных экземпляров вы ссылаетесь. Рассмотрим листинг 2.14. При выполнении этого кода выбрасывается исключение, в котором Spring просит точно указать, какой именно из экземпляров вы хотите использовать.

Листинг 2.14. Ссылка на экземпляр Parrot по типу

```

public class main {

    public static void main(String[] args) {
        var context = new
            AnnotationConfigApplicationContext(ProjectConfig.class);

        Parrot p = context.getBean(Parrot.class); ←
        System.out.println(p.getName());
    }
}

```

В этой строке мы получим исключение, так как Spring не может догадаться, на какой из трех экземпляров Parrot вы ссылаетесь

При выполнении приложения получим исключение, подобное представленному в следующем примере кода.

```

Exception in thread "main"
org.springframework.beans.factory.NoUniqueBeanDefinitionException:
No
qualifying bean of type 'main.Parrot' available: expected single
matching

```

```
bean but found 3:  
parrot1,parrot2,parrot3 ← Имена бинов Parrot в контексте  
at ...
```

Чтобы устранить эту неоднозначность, необходимо точно сослаться на один из экземпляров по имени его бина. По умолчанию Spring использует в качестве названий бинов имена методов с аннотациями `@Bean`. Напомню: именно поэтому мы отказались от глаголов. В нашем случае имена бинов — это `parrot1`, `parrot2` и `parrot3` (напомню, что каждый метод соответствует отдельному бину). Как вы, вероятно, заметили, эти названия упоминаются в предыдущем примере кода, в сообщении об исключении. Изменим метод `main` таким образом, чтобы он явно ссылался на один из этих бинов по имени. Посмотрите, как я обращаюсь на бин `parrot2`, в листинге 2.15.

Листинг 2.15. Ссылка на бин по идентификатору

```
public class main {  
  
    public static void main(String[] args) {  
        var context = new  
            AnnotationConfigApplicationContext(ProjectConfig.class);  
  
        Parrot p = context.getBean("parrot2", Parrot.class); ← Первый параметр — это  
        System.out.println(p.getName());                                имя экземпляра, на  
    }                                                               который мы ссылаемся  
}
```

Теперь мы больше не получим исключение при выполнении приложения. Вместо этого в консоль будет выведено имя второго попугая — `Miki`.

Если вы захотите присвоить бину другое имя, то можете указать его в аннотации `@Bean` в качестве атрибута `name` или `value`. Любой из следующих вариантов кода меняет имя бина на `miki`:

- `@Bean(name = "miki");`
- `@Bean(value = "miki");`
- `@Bean("miki").`

В следующем примере показано, как это изменение повлияет на код. Если вы захотите проверить его работу, то найдете его в проекте `sq-ch2-ex4`:

```
@Bean(name = "miki") ← Имя бина  
Parrot parrot2() {  
    var p = new Parrot();  
    p.setName("Miki"); ← Кличка попугая  
    return p;  
}
```

ОПРЕДЕЛЕНИЕ БИНА В КАЧЕСТВЕ ПЕРВИЧНОГО

Как уже говорилось ранее, в контексте Spring можно использовать несколько бинов одного типа, но при этом на них необходимо ссылаться по именам. Есть и другой способ сослаться на бин в подобном случае.

При наличии в контексте Spring нескольких бинов одного типа один из бинов можно сделать *первичным*. Для этого используется аннотация `@Primary`. Spring применяет первичный бин тогда, когда существует несколько бинов данного типа и конкретное имя не указано. По сути, первичный бин — это просто бин, который Spring использует по умолчанию. В следующем примере показано, как выглядит метод для бина, выбранного в качестве первичного:

```
@Bean
@Primary
Parrot parrot2() {
    var p = new Parrot();
    p.setName("Miki");
    return p;
}
```

Если теперь сослаться на экземпляр `Parrot` без указания имени, то Spring по умолчанию выберет `Miki`. Естественно, первичным может быть только один бин. Данный пример реализован в проекте `sq-ch2-ex5`.

2.2.2. Добавление бинов в контекст Spring с помощью стереотипных аннотаций

Есть другой способ добавления бинов в контекст Spring (позже мы сравним разные способы и обсудим, в каких случаях какой из них следует использовать). Напомню: нам необходимо включать бины в контекст Spring, потому что именно так мы сообщаем Spring, какими экземплярами объектов приложения должен управлять фреймворк. В Spring есть и другие способы добавления бинов в контекст. Вы скоро поймете, что выбор наиболее удобного из них зависит от конкретной ситуации. Например, в случае стереотипных аннотаций для включения бинов в контекст Spring можно обойтись меньшим количеством кода.

Позже вы узнаете, что в Spring есть несколько стереотипных аннотаций. Но в данном пункте я обращаю ваше внимание только на общий принцип их использования. Мы возьмем простейшую из них, `@Component`, и воспользуемся ею в наших примерах. Стереотипные аннотации размещаются над классом, экземпляр которого мы хотим добавить в контекст Spring. Таким образом мы сообщаем Spring, что данный класс является компонентом. Когда приложение создает контекст Spring, фреймворк, в свою очередь, создает экземпляр класса, отмеченного как компонент, и добавляет этот экземпляр в свой контекст. При использовании

данного варианта нам также понадобится класс конфигурации, поскольку в нем мы сообщаем Spring, где искать классы со стереотипными аннотациями. Более того, в одном приложении можно применять оба способа (со стереотипными аннотациями и аннотациями `@Bean`) — эти более сложные примеры мы рассмотрим в последующих главах.

Чтобы добавить стереотипную аннотацию, нужно сделать следующее (рис. 2.12).

1. Отметить аннотацией `@Component` те классы, экземпляры которых вы хотите поместить в контекст Spring (в нашем случае это класс `Parrot`).
2. Используя аннотацию `@ComponentScan` в классе конфигурации, сообщить Spring, где находятся классы, отмеченные аннотацией `@Component`.

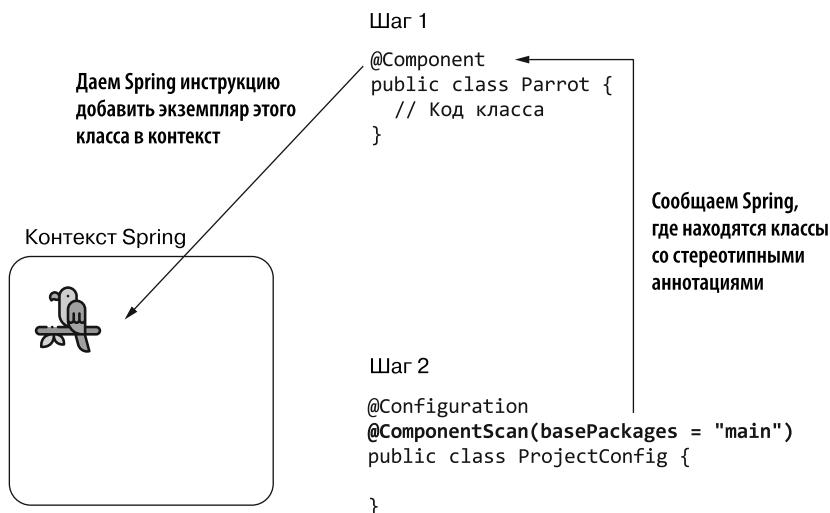


Рис. 2.12. Применение стереотипных аннотаций происходит в два этапа. Вначале с помощью стереотипной аннотации (`@Component`) мы отмечаем класс, бин которого следует добавить в контекст Spring. Затем с помощью аннотации `@ComponentScan` мы сообщаем фреймворку, где находятся классы со стереотипными аннотациями

Рассмотрим эту процедуру на примере нашего класса `Parrot`. Чтобы добавить экземпляр этого класса в контекст Spring, нужно снабдить класс `Parrot` одной из стереотипных аннотаций, например `@Component`.

В листинге 2.16 продемонстрировано использование аннотации `@Component` на примере класса `Parrot`. Код этого примера находится в проекте `sq-ch2-ex6`.

Но постойте! Ведь этот код не работает. По умолчанию Spring не ищет классы со стереотипными аннотациями. Поэтому, если оставить все как есть, фреймворк

не добавит бин класса `Parrot` в свой контекст. Чтобы Spring искал классы со стереотипными аннотациями, нужно воспользоваться аннотацией `@ComponentScan` в классе конфигурации проекта. В моем случае имя пакета — `main` (листинг 2.17).

Листинг 2.16. Применение стереотипной аннотации к классу Parrot

```
@Component ←
public class Parrot {
    private String name;

    public String getName() {
        return name;
    }

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

Поставив перед классом аннотацию `@Component`, мы даем Spring команду создать экземпляр этого класса и добавить его в контекст

Листинг 2.17. Использование аннотации `@ComponentScan`, чтобы сообщить Spring, где находятся классы со стереотипными аннотациями

```
@Configuration
@ComponentScan(basePackages = "main") ←
public class ProjectConfig {
}
```

Используя в аннотации атрибут `basePackages`, мы указываем Spring, где находятся классы со стереотипными аннотациями

Теперь мы сообщили Spring следующее.

1. Экземпляры каких классов нужно добавить в контекст (`Parrot`).
2. Где находятся эти классы (с помощью аннотации `@ComponentScan`).

ПРИМЕЧАНИЕ

Теперь нам не нужны специальные методы для определения бинов. Данный способ выглядит лучше, чем предыдущий: ведь мы получили тот же результат, но написали меньше кода. Однако не спешите делать выводы, сначала дочитайте главу до конца. Вы узнаете, что оба способа хороши в зависимости от конкретной ситуации.

Чтобы убедиться, что Spring создает бин и добавляет его в контекст, можно переписать метод `main` следующим образом (листинг 2.18).

Листинг 2.18. Тестирование конфигурации Spring в методе main

```
public class main {
    Public static void main(string[] args) {
        var context = new
            AnnotationConfigApplicationContext(ProjectConfig.class);
```

```

Parrot p = context.getBean(Parrot.class);
System.out.println(p);           ← Выводит в консоль стандартное представление
                                ← экземпляра String, полученное из контекста Spring
System.out.println(p.getName()); ← Выводит в консоль null, поскольку мы не дали кличку
                                ← этому попугаю, добавленному в контекст Spring
}
}

```

Запустив приложение, вы увидите, что Spring добавляет в контекст экземпляр класса `Parrot`, так как первое из выведенных в консоль значений является стандартным представлением экземпляра `String`. Но второе выведенное значение — `null`, поскольку мы не дали кличку этому попугаю. Spring всего лишь создает экземпляр класса, наша задача — изменить этот экземпляр так, как нам нужно (например, дать попугаю кличку).

Теперь, когда мы освоили два наиболее часто встречающихся способа добавления бинов в контекст Spring, сделаем их краткое сравнение (табл. 2.1).

Используя Spring в реальных приложениях, вы скоро заметите, что применяете стереотипные аннотации везде, где только можно (ведь они требуют меньше кода), а `@Bean` — лишь когда нет других вариантов (например, когда вы создаете бин для класса, являющегося частью библиотеки, и не можете изменить этот класс, чтобы добавить в него стереотипную аннотацию).

Таблица 2.1. Преимущества и недостатки: сравнительная характеристика двух способов добавления бинов в контекст Spring, которая позволит вам выбрать, когда использовать каждый из них

Аннотация <code>@Bean</code>	Стереотипные аннотации
<p>1. Вы полностью контролируете создание экземпляра, который добавляете в контекст Spring. Вы сами создаете экземпляр и описываете его конфигурацию в теле метода, сопровождаемого аннотацией <code>@Bean</code>. Spring берет этот экземпляр и добавляет его в контекст, ничего не меняя.</p> <p>2. Благодаря данному методу Spring может добавлять в контекст несколько экземпляров одного типа. Как вы помните, именно так мы включили в контекст Spring три экземпляра класса <code>Parrot</code> в подразделе 2.1.1.</p> <p>3. С помощью аннотации <code>@Bean</code> можно добавить в контекст Spring экземпляр любого объекта. Класс, описывающий этот объект, не обязан определяться в приложении: как вы помните, мы так включили в контекст Spring экземпляры типа <code>String</code> и <code>Integer</code>.</p> <p>4. Для каждого бина, добавляемого в контекст Spring, приходится писать отдельный метод, что увеличивает количество шаблонного кода в приложении. Поэтому в наших проектах <code>@Bean</code> будет использоваться как запасной вариант там, где стереотипные аннотации не подходят</p>	<p>1. Вы получаете контроль над экземпляром только после того, как фреймворк его создаст.</p> <p>2. Таким образом можно добавить в контекст только один экземпляр класса.</p> <p>3. С помощью стереотипных аннотаций можно создавать бины только для классов приложения. Например, не получится загрузить бин типа <code>String</code> или <code>Integer</code>, как мы сделали в подразделе 2.1.1, с помощью аннотации <code>@Bean</code>, поскольку эти классы не принадлежат приложению и мы не можем их изменить, добавив стереотипную аннотацию.</p> <p>4. При добавлении бинов в контекст Spring посредством стереотипных аннотаций в приложении не появляется новый шаблонный код. Как правило, этот способ более предпочтителен, если класс принадлежит приложению</p>

ИСПОЛЬЗОВАНИЕ @POSTCONSTRUCT ДЛЯ УПРАВЛЕНИЯ СОЗДАННЫМ ЭКЗЕМПЛЯРОМ

Как уже говорилось, используя стереотипную аннотацию, мы даем Spring команду создать бин и добавить его в контекст. Но, в отличие от аннотации @Bean, в этом случае мы не можем полностью контролировать создание экземпляра. Используя @Bean, мы могли определить имя каждого экземпляра Parrot, который включался в контекст Spring, однако при использовании @Component у нас нет возможности сделать что-либо после того, как Spring вызвал конструктор класса Parrot. Как быть, если нужно выполнить несколько инструкций сразу после создания бина? Для этого можно воспользоваться аннотацией @PostConstruct.

Spring позаимствовал аннотацию @PostConstruct из Java EE. Ее также можно применять к бинам Spring, чтобы определить несколько инструкций, которые Spring выполнит после создания бина. Для этого нужно определить метод в классе компонента и снабдить этот метод аннотацией @PostConstruct. Spring вызовет метод с этой аннотацией после того, как закончится выполнение конструктора.

Добавим в файл pom.xml проекта Maven зависимость, необходимую для использования аннотации @PostConstruct:

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

Для версий, предшествующих Java 11, эту зависимость добавлять не нужно. До Java 11 все зависимости Java EE входили в состав JDK. Но в Java 11 JDK был очищен от всех API, не имеющих отношения к SE, в том числе от зависимостей Java EE. Чтобы использовать функционал из удаленных API (такой как @PostConstruct), нужно явно добавить в приложение соответствующую зависимость.

Теперь можно определить в классе Parrot следующий метод:

```
@Component
public class Parrot {

    private String name;

    @PostConstruct
    public void init() {
        this.name = "Kiki";
    }

    // Какой-то код
}
```

Данный пример вы найдете в проекте sq-ch2-ex7. Если теперь ввести в консоли кличку попугая, то приложение выведет значение Kiki.

Аналогичным образом, хоть и реже встречающимся в реальных приложениях, можно применять аннотацию @PreDestroy. Этой аннотацией отмечают метод, который Spring выполняет непосредственно перед закрытием и очисткой контекста. Аннотация @PreDestroy также описана в JSR-250, откуда ее и позаимствовал Spring. Однако я советую разработчикам по возможности воздерживаться от ее использования и искать другие способы выполнить что-либо перед тем, как Spring очистит контекст, — главным образом потому, что попытка очистить контекст Spring может оказаться неудачной. Предположим, вы определили в методе с @PreDestroy какое-либо важное действие (например, закрытие соединения с базой данных). Если Spring не вызовет этот метод — возможны крупные неприятности.

2.2.3. Программное добавление бинов в контекст Spring

Здесь мы рассмотрим программное добавление бинов в контекст Spring. Возможность программного добавления бинов в контекст Spring появилась в Spring 5, что значительно повысило гибкость фреймворка, поскольку позволило включать в контекст новые экземпляры непосредственно, вызывая метод экземпляра контекста. Данный способ используется в тех случаях, когда нужно реализовать нестандартное добавление бинов в контекст и возможностей @Bean и стереотипных аннотаций для этого недостаточно. Предположим, нам нужно зарегистрировать в контексте Spring те или иные бины, в зависимости от специфической конфигурации приложения. @Bean и стереотипные аннотации позволяют реализовать большинство сценариев, но не такой, например, код:

```
if (condition) {
    registerBean(b1); ← Если условие истинно, добавить в контекст Spring некий бин
} else {
    registerBean(b2); ← Иначе: добавить в контекст Spring другой бин
}
```

Продолжая наш пример с попугаями, рассмотрим следующий сценарий: приложение считывает коллекцию попугаев. Некоторые из них зеленые, остальные оранжевые. Мы хотим, чтобы приложение добавляло в контекст Spring только зеленых попугаев (рис. 2.13).

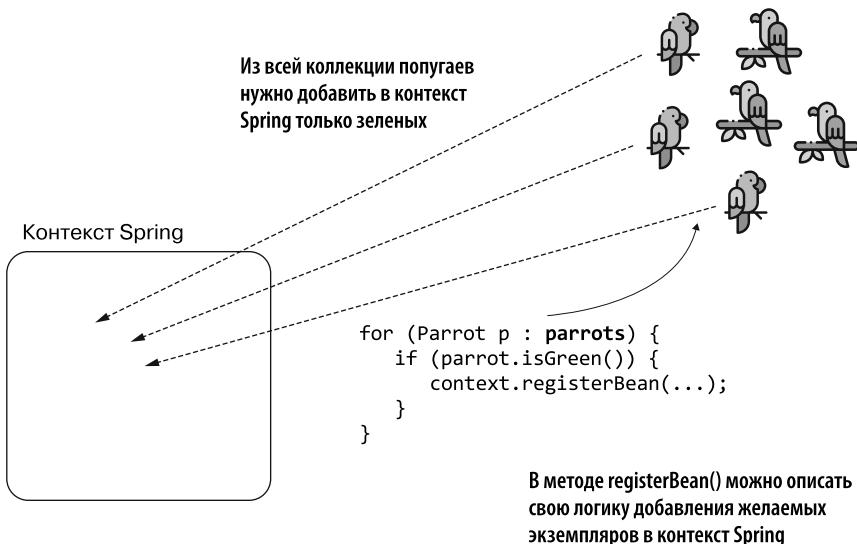


Рис. 2.13. Метод registerBean() позволяет добавлять в контекст Spring выбранные экземпляры объектов

Посмотрим, как работает этот метод. Чтобы программно добавить бин в контекст Spring, достаточно просто вызвать метод `registerBean()` экземпляра `ApplicationContext`. Как показано в следующем фрагменте кода, метод `registerBean()` принимает четыре параметра.

```
<T> void registerBean(
    String beanName,
    Class<T> beanClass,
    Supplier<T> supplier,
    BeanDefinitionCustomizer... customizers);
```

1. Первый параметр, `beanName`, — это имя бина, добавляемого в контекст Spring. Если вы не хотите присвоить имя бину, который включается в контекст, при вызове метода присвойте этому параметру значение `null`.
2. Второй параметр — класс, который определяет бин, добавляемый в контекст. Если вы хотите добавить экземпляр класса `Parrot`, то значением этого параметра будет `Parrot.class`.
3. Третий параметр — это экземпляр `Supplier`. Реализация `Supplier` нужна, чтобы возвращать значение экземпляра, добавляемого в контекст. Напомню: `Supplier` — это функциональный интерфейс, который входит в пакет `Java.util.function`. Назначение реализации `Supplier` состоит в том, чтобы возвращать заданное значение, не принимая параметров.

4. Четвертый, и последний, параметр — это аргумент переменной длины (`varargs`) `BeanDefinitionCustomizer`. (Если вы впервые слышите это слово, ничего страшного: `BeanDefinitionCustomizer` — просто еще один интерфейс, который используется для настройки различных свойств бина, например, чтобы сделать бин первичным.) Определив его как аргумент переменной длины, мы можем либо совсем пропустить этот параметр, либо присвоить ему несколько значений типа `BeanDefinitionCustomizer`.

В проекте `sq-ch2-ex8` вы найдете пример использования метода `registerBean()`. Там вы увидите, что класс конфигурации проекта пуст, а класс `Parrot`, который мы использовали как пример определения бина, представляет собой старый добрый объект Java (POJO), и аннотации для него не используются. В следующем фрагменте кода показан класс конфигурации, созданный для этого примера:

```
@Configuration
public class ProjectConfig {
}
```

Класс `Parrot` для создания бина я определил так:

```
public class Parrot {
    private String name;
    // Здесь находятся геттеры и сеттеры
}
```

Чтобы добавить экземпляр типа `Parrot` в контекст Spring, я использовал метод `registerBean()` в главном методе проекта. Код метода `main` представлен в листинге 2.19. На рис. 2.14 проиллюстрирован синтаксис вызова метода `registerBean()`.

Листинг 2.19. Добавление бина в контекст Spring с помощью метода `registerBean()`

```
public class main {
    public static void main(String[] args) {
        var context =
            new AnnotationConfigApplicationContext(
                ProjectConfig.class);
```

Создание экземпляра, который будет добавлен в контекст Spring

```
        Parrot x = new Parrot(); ←
        x.setName("Kiki");
```

Определение Supplier, который будет возвращать этот экземпляр

```
        Supplier<Parrot> parrotSupplier = () -> x; ←
```

```
        context.registerBean("parrot1",
            Parrot.class, parrotSupplier); ←
```

Вызов метода registerBean(), добавляющего экземпляр в контекст Spring

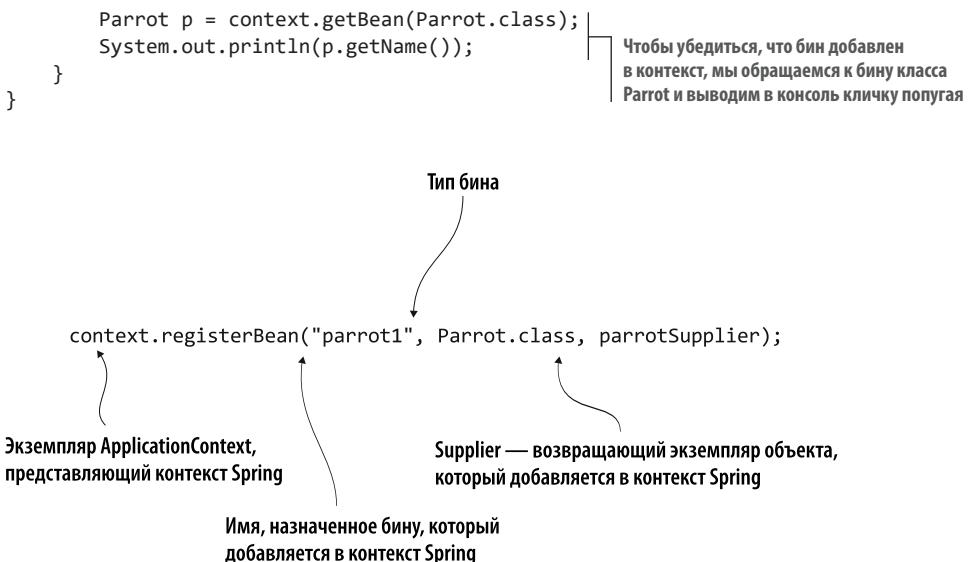


Рис. 2.14. Вызов метода `registerBean()`, чтобы программно добавить бин в контекст Spring

Один или несколько экземпляров конфигураторов бина можно использовать как последние параметры, чтобы задать добавляемому бину различные характеристики. Например, можно сделать бин первичным, изменив вызов метода `registerBean()`, как показано в следующем примере. Первичный бин — это экземпляр, который Spring выбирает по умолчанию при наличии в контексте нескольких бинов одного типа:

```

context.registerBean("parrot1",
    Parrot.class,
    parrotSupplier,
    bc -> bc.setPrimary(true));
    
```

Вы только что сделали первый большой шаг в мир Spring. Умение добавлять бины в контекст Spring может показаться не особо великим знанием, однако оно гораздо важнее, чем кажется на первый взгляд. Освоив этот навык, вы сможете ссылаться на бины, размещенные в контексте Spring, о чём мы поговорим в главе 3.

ПРИМЕЧАНИЕ

В настоящей книге используются только современные подходы к конфигурации. Однако я считаю важным также рассказать, как разработчики настраивали фреймворк в первое время существования Spring. Тогда для написания конфигураций фреймворка мы использовали XML. В приложении Б приводится краткий пример, который дает представление о том, какой XML вам бы пришлось написать, чтобы добавить бин в контекст Spring.

РЕЗЮМЕ

- Первое, что нужно освоить в Spring, — это добавление в контекст Spring экземпляров объектов (называемых бинами). Контекст Spring можно представить как ведро, в которое помещают экземпляры, чтобы затем Spring мог ими управлять. Spring «видит» только те экземпляры, которые были добавлены в контекст.
- Есть три способа добавления бинов в контекст Spring: с помощью аннотации `@Bean` и стереотипных аннотаций, а также программно.
 - Посредством аннотации `@Bean` в контекст Spring в качестве бина можно добавить экземпляр (или даже несколько экземпляров) объекта любого типа. Такой подход является более гибким, чем стереотипные аннотации. Однако он требует написания большего кода, поскольку необходимо создавать в классе конфигурации отдельный метод для каждого независимого экземпляра, включаемого в контекст.
 - С помощью стереотипных аннотаций можно создавать бины только для классов, определенных в приложении. (Для этого применяются специфические аннотации, такие как `@Component`.) При подобном подходе требуется писать меньше кода, благодаря чему конфигурацию легче читать. Данный способ предпочтительнее, чем `@Bean`, для тех классов, которые вы создали сами и можете снабдить аннотациями.
 - Метод `registerBean()` позволяет реализовать собственную логику добавления бинов в контекст Spring. Напомню: этот метод можно использовать только в Spring 5 или в более поздних версиях.

3

Контекст Spring: создаем новые бины

В этой главе

- ✓ Как устанавливать связи между бинами.
- ✓ Что такое внедрение зависимостей.
- ✓ Как получить доступ к бинам через контекст Spring посредством внедрения зависимостей.

В главе 2 вы изучили контекст Spring — место в памяти приложения, куда помещаются экземпляры объектов, управляемые фреймворком. Поскольку Spring действует по принципу IoC, рассмотренного в главе 1, необходимо сообщить Spring о том, какие из объектов приложения он должен контролировать, чтобы расширить их функционал за счет возможностей фреймворка. В главе 2 мы рассмотрели несколько способов, позволяющих добавить экземпляр объекта в контекст Spring. Вы также узнали, что эти экземпляры (бины) добавляются в контекст Spring для того, чтобы сообщить Spring об их существовании.

В настоящей главе вы узнаете, как получить доступ к бинам, добавленным в контекст Spring. В главе 2 мы использовали для этого метод `getBean()` экземпляра контекста. Но в реальных приложениях приходится обращаться из одного бина к другому напрямую. Для этого Spring может предоставлять ссылку на экземпляр, размещенный в контексте фреймворка. Таким образом устанавливаются

связи между бинами (один бин получает ссылку на другой, и после этого он может делегировать ему вызовы). Как вы, вероятно, уже знаете, в объектно-ориентированном программировании при реализации поведения объектов часто возникает необходимость в делегировании определенных функций другим объектам. Поэтому при использовании фреймворка Spring тоже необходимо уметь устанавливать такие связи между объектами.

Вы скоро узнаете, что есть еще много других способов получить доступ к объектам, добавленным в контекст Spring. Мы изучим все эти варианты на примерах, иллюстрациях и, разумеется, на фрагментах кода. В конце главы вам придется применить освоенные навыки использования контекста Spring, описания конфигурации бинов и установки связей между бинами. Это основа работы со Spring; нет ни одного приложения Spring, в котором бы не применялись указанные в данной главе приемы. Именно поэтому все, что есть в настоящем издании (и все, что вы узнаете из других книг, статей и обучающих видеоматериалов), опирается на глубокое понимание базы, показанной в главах 2–5.

В главе 2 вы научились добавлять бины в контекст Spring с помощью аннотации `@Bean`. Мы начнем раздел 3.1 с того, что установим связи между двумя бинами, описанными в классе конфигурации с помощью `@Bean`. Рассмотрим следующие два способа:

- установить связь между двумя бинами, непосредственно вызывая методы, которые создают эти бины (так называемый *монтаж – wiring*);
- настроить Spring так, чтобы он предоставлял значение через параметр метода (*автомонтаж – auto-wiring*).

Затем, в разделе 3.2, мы рассмотрим третий способ — технологию, существующую благодаря принципу IoC, которая называется *внедрением зависимостей* (dependency injection, DI). Мы рассмотрим, как использовать DI в Spring, применяя аннотацию `@Autowired` для установки связей между двумя бинами (что также является примером автомонтажа). В реальных проектах вы будете использовать оба способа.

ПРИМЕЧАНИЕ

Может показаться, что примеры, представленные в главах 2 и 3, слишком далеки от реальных условий. В конце концов, в настоящих приложениях не бывает попугаев и их владельцев! Но я хочу, чтобы вы погружались в предмет постепенно, начиная с самого простого, и фокусировались на тех важных элементах синтаксиса, которые впоследствии вы будете использовать практически в любом Spring-приложении. Тогда я буду уверен, что вы хорошо понимаете, каким образом работают изучаемые приемы, и концентрируете внимание только на них. Начиная с главы 4, наши учебные разработки станут ближе к тому, что вам встретится в реальных проектах.

3.1. УСТАНОВКА СВЯЗЕЙ МЕЖДУ БИНАМИ, ОПИСАННЫМИ В ФАЙЛЕ КОНФИГУРАЦИИ

Далее вы научитесь устанавливать связи между двумя бинами, описанными в классе конфигурации посредством методов с аннотациями `@Bean`. Такой способ работы с бинами будет часто встречаться вам в реальных приложениях. В главе 2 мы выяснили, что аннотация `@Bean` применяется для добавления бинов в контекст Spring в тех случаях, когда невозможно изменить класс, для которого создается бин, — например, если класс входит в состав JDK или другой зависимости. Для установки связей между такими бинами нужно освоить методы, описанные в данном разделе. Мы рассмотрим, как работают эти методы; затем я покажу вам последовательность действий, которую нужно выполнить, чтобы установить связь между бинами; далее мы применим полученные знания в нескольких небольших проектах.

Представим, что в контексте Spring есть два экземпляра: попугай и человек. Мы создадим эти экземпляры и вставим их в контекст. Мы хотим сделать человека владельцем попугая — для этого данные экземпляры нам нужно связать. Этот простой пример поможет нам изучить два способа установки связи между бинами в контексте Spring, ничего не усложняя и концентрируя внимание только на конфигурации Spring.

Итак, любая установка взаимосвязей между бинами (и монтаж, и автомонтаж) выполняется в два этапа (рис. 3.1).

1. Сначала нужно добавить бины человека и попугая в контекст Spring (этому вы научились в главе 2).
2. Затем следует установить взаимосвязь между человеком и попугаем.

На рис. 3.2 более схематично изображена взаимосвязь типа *has-A* между объектами «человек» и «попугай».

Прежде чем углубиться в изучение названных способов установки связи между бинами, рассмотрим первый пример этой главы (проект `sq-ch3-ex1`), чтобы вспомнить, как добавлять бины в контекст Spring с помощью методов с аннотацией `@Bean` (см. подраздел 2.2.1 (шаг 1)). Мы добавим два экземпляра, «попугай» и «человек», а затем установим связь между ними (шаг 2). В подразделе 3.1.1 мы реализуем монтаж для методов с аннотацией `@Bean`, а в подразделе 3.1.2 — автомонтаж. В следующем фрагменте кода показано, как добавить в файл `rom.xml` проекта Maven зависимость для контекста Spring:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.7.RELEASE</version>
</dependency>
```

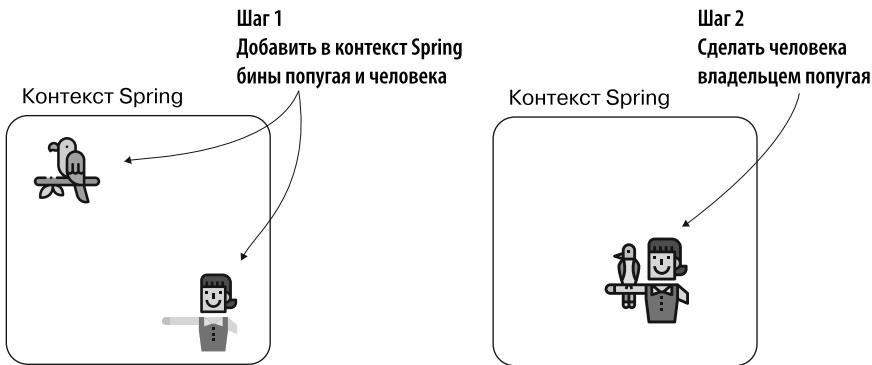


Рис. 3.1. В контексте Spring есть два бина. Мы хотим установить взаимосвязь между ними, чтобы затем один объект мог делегировать другому выполнение своих функций. Для этого можно использовать монтаж (то есть установку связи между бинами путем прямого вызова методов, которые объявляют эти бины) или же автомонтаж. Мы будем применять внедрение зависимостей — возможность, предоставляемую фреймворком

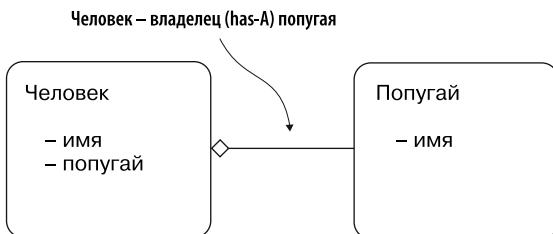


Рис. 3.2. Установка связи между бинами. Эта упрощенная диаграмма отражает взаимосвязь has-A (владение) между объектами «человек» и «попугай». Мы будем устанавливать такую связь посредством монтажа и автомонтажа

Затем мы определим класс `Parrot`, описывающий объект «попугай», и класс `Person`, описывающий объект «человек». Определение класса `Parrot` представлено в следующем фрагменте кода:

```
public class Parrot {
    private String name;

    // геттеры и сеттеры

    @Override
    public String toString() {
        return "Parrot : " + name;
    }
}
```

В следующем фрагменте кода содержится определение класса Person:

```
public class Person {  
  
    private String name;  
    private Parrot parrot;  
  
    // геттеры и сеттеры  
}
```

В листинге 3.1 показано, как с помощью аннотации @Bean определить два бина в классе конфигурации.

Листинг 3.1. Определение бинов Person и Parrot

```
@Configuration  
public class ProjectConfig {  
  
    @Bean  
    public Parrot parrot() {  
        Parrot p = new Parrot();  
        p.setName("Koko");  
        return p;  
    }  
  
    @Bean  
    public Person person() {  
        Person p = new Person();  
        p.setName("Ella");  
        return p;  
    }  
}
```

Теперь можно определить класс Main и убедиться, что экземпляры «человек» и «попугай» все еще не связаны между собой (листинг 3.2).

Листинг 3.2. Определение класса Main

```
public class Main {  
  
    public static void main(String[] args) {  
        var context = new AnnotationConfigApplicationContext(  
            ProjectConfig.class);  
  
        Person person =  
            context.getBean(Person.class);  
  
        Parrot parrot =  
            context.getBean(Parrot.class);  
  
        System.out.println(  
            "Person's name: " + person.getName());  
    }  
}
```

Создаем экземпляр контекста Spring
на основе файла конфигурации

Получаем из контекста Spring
ссылку на бин Person

Получаем из контекста Spring
ссылку на бин Parrot

Выводим в консоль имя человека, чтобы
убедиться, что бин Person есть в контексте

```

System.out.println(
    "Parrot's name: " + parrot.getName()); ← Выводим в консоль имя попугая, чтобы
                                            убедиться, что бин Parrot есть в контексте

System.out.println(
    "Person's parrot: " + person.getParrot()); ← Выводим в консоль Person's parrot
                                                («попугай принадлежит человеку»)
                                                и убеждаемся, что связь между
                                                данными объектами еще
                                                не установлена

}
  
```

Запустив приложение, вы увидите в консоли примерно следующее:

Person's name: Ella	В контексте Spring есть бин Person
Parrot's name: Koko	В контексте Spring есть бин Parrot
Person's parrot: null	Связь между бинами Person и Parrot не установлена

Главное, что нужно здесь заметить, — несмотря на то что экземпляры `Person` и `Parrot` есть в контексте, при попытке вывести в консоль `Person's parrot` («попугай принадлежит человеку») получаем `null`. Вывод `null` означает, что связь между этими экземплярами еще не установлена (рис. 3.3).



Рис. 3.3. Мы добавили в контекст два бина. Теперь нужно описать связь между ними

3.1.1. Монтаж бинов путем прямого вызова одного метода с аннотацией @Bean из другого такого же метода

Установим связь между двумя экземплярами — `Person` и `Parrot`. Первый способ (монтаж) заключается в вызове одного метода из другого в классе конфигурации. Он очень простой и поэтому популярный. В листинге 3.3 вы заметите небольшое изменение, которое я внес в класс конфигурации, чтобы установить связь между человеком и попугаем (рис. 3.4). Чтобы каждая операция хранилась отдельно и вам было легче разбирать код, я сохранил новый вариант в особом проекте — `sq-ch3-ex2`.

Листинг 3.3. Установка связи между бинами путем прямого вызова метода

```
@Configuration
public class ProjectConfig {

    @Bean
    public Parrot parrot() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }

    @Bean
    public Person person() {
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot()); ← Создание ссылки из бина person на бин parrot
        return p;
    }
}
```

Теперь, запустив приложение, вы увидите, что текст в консоли изменился. Вы обнаружите (см. следующий пример кода), что во второй строке `Ella` (экземпляр «человек» в контексте Spring) является хозяином `Koko` (экземпляром «попугай» в контексте Spring):

```
Person's name: Ella
Person's parrot: Parrot : Koko
```

Установливаем связь между бинами person и parrot путем прямого вызова метода, который возвращает необходимый нам бин

```
@Configuration
public class ProjectConfig {

    @Bean
    public Parrot parrot() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }

    @Bean
    public Person person() {
        Person p = new Person();
        p.setName("Ella"); ←
        p.setParrot(parrot());
        return p;
    }
}
```

В результате между бинами person и parrot устанавливается связь типа has-A: человек является владельцем попугая

Рис. 3.4. Устанавливаем связь между бинами путем прямого монтажа.

Данный способ подразумевает вызов метода, возвращающего бин, с которым устанавливается прямая связь. Этот метод нужно вызывать из другого метода — определяющего бин, для которого устанавливается зависимость

Когда я объясняю этот способ студентам, кто-нибудь обязательно задает вопрос: не создается ли при этом два экземпляра типа `Parrot` (рис. 3.5) — один при добавлении в контекст Spring и еще один, когда метод `person()` напрямую вызывает метод `parrot()`? Нет. На самом деле в приложении существует только один экземпляр `parrot`.

На первый взгляд это может показаться странным, но Spring достаточно «умен», чтобы понять, что вы, вызывая метод `parrot()`, хотите сослаться на размещенный в контексте бин `parrot`. Если для определения бина в контексте Spring была использована аннотация `@Bean`, то Spring отслеживает вызываемые методы и в конкретной ситуации может применить ту или иную логику (о том, как именно Spring вмешивается в выполнение методов, вы узнаете в главе 6). Пока что просто запомните, что, когда метод `person()` вызывает метод `parrot()`, Spring применяет некую логику, которая будет описана далее.

1. Spring вызывает метод `parrot()`, чтобы создать бин `parrot` и добавить его в контекст

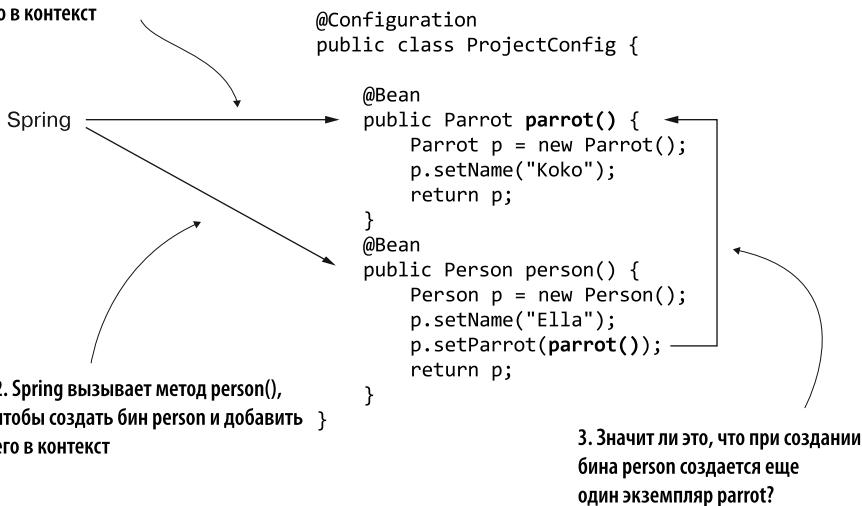


Рис. 3.5. При первом вызове метода `parrot()` с аннотацией `@Bean` Spring создает экземпляр `parrot`. Затем Spring создает экземпляр `person`, вызывая метод `person()` — еще один с аннотацией `@Bean`. Метод `person()`, в свою очередь, непосредственно вызывает первый метод, `parrot()`. Значит ли это, что создается два экземпляра типа `Parrot`?

Если в контексте уже есть бин `parrot`, то вместо того, чтобы вызвать метод `parrot()`, Spring сразу извлекает этот экземпляр из контекста. Если же бин `parrot` в контексте еще не создан, то Spring вызывает метод `parrot()` и возвращает созданный бин (рис. 3.6).

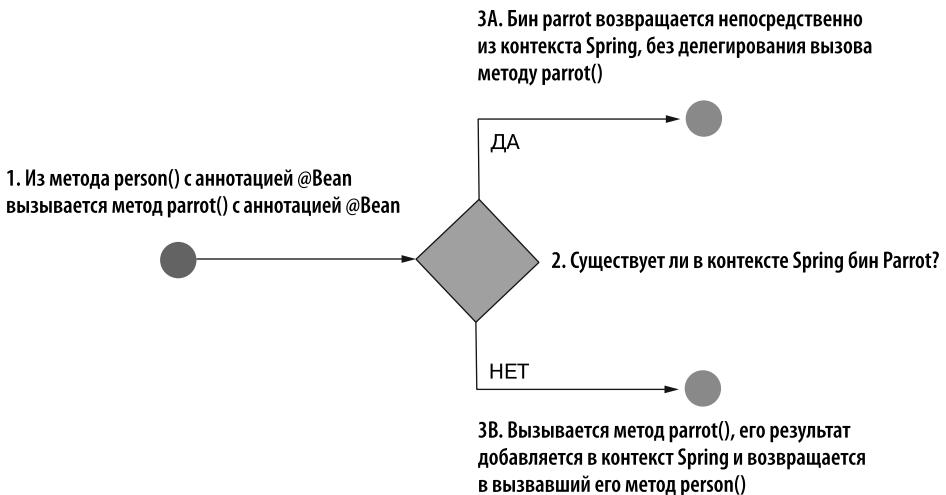


Рис. 3.6. Когда один метод с аннотацией `@Bean` вызывает другой метод с аннотацией `@Bean`, Spring понимает, что вы хотите создать связь между бинами. Если такой бин уже существует (вариант 3A), Spring возвращает его, не передавая вызов методу с `@Bean`. Если же такого бина еще нет (вариант 3B), то Spring создает его и возвращает на него ссылку

В сущности, мы можем легко проверить, так ли это: достаточно добавить в класс `Parrot` конструктор без аргументов и вывести из данного конструктора сообщение в консоль. Сколько раз сообщение появится в консоли? Если все верно, то мы увидим сообщение только один раз. Проведем эксперимент. В следующем фрагменте кода я добавил в класс `Parrot` конструктор без аргументов:

```

public class Parrot {

    private String name;

    public Parrot() {
        System.out.println("Parrot created");
    }

    // геттеры и сеттеры

    @Override
    public String toString() {
        return "Parrot : " + name;
    }
}

```

Снова запустив приложение, мы увидим, что его результат изменился (см. ниже) — теперь там появляется сообщение `Parrot created` (Попугай создан). Вы заметите,

что оно выводится только один раз, а следовательно, Spring управляет созданием бинов и вызывает метод `parrot()` лишь однажды:

```
Parrot created
Person's name: Ella
Person's parrot: Parrot : Koko
```

3.1.2. Монтаж бинов путем передачи параметра в метод с аннотацией @Bean

Существует и другой способ прямого вызова метода с аннотацией `@Bean`. Вместо того чтобы непосредственно вызывать метод, определяющий бин, на который мы хотим сослаться, мы будем добавлять параметр к методу соответствующего типа объектов, и Spring будет предоставлять нам значение через этот параметр (рис. 3.7). Данный способ более гибкий, чем тот, что был описан в подразделе 3.1.1. Теперь не имеет значения, как был определен бин, на который мы ссылаемся: с помощью аннотации `@Bean` или посредством стереотипной аннотации наподобие `@Component` (см. главу 2). Однако, как показывает мой опыт, не только гибкость привлекает разработчиков, использующих настоящий подход: то, какой из вариантов выбрать при работе с бинами, зависит главным образом от личных предпочтений разработчика. Не могу сказать, что какой-то из способов лучше другого — вы скоро заметите, что оба они широко применяются в реальных приложениях, так что необходимо понять и научиться использовать оба варианта.

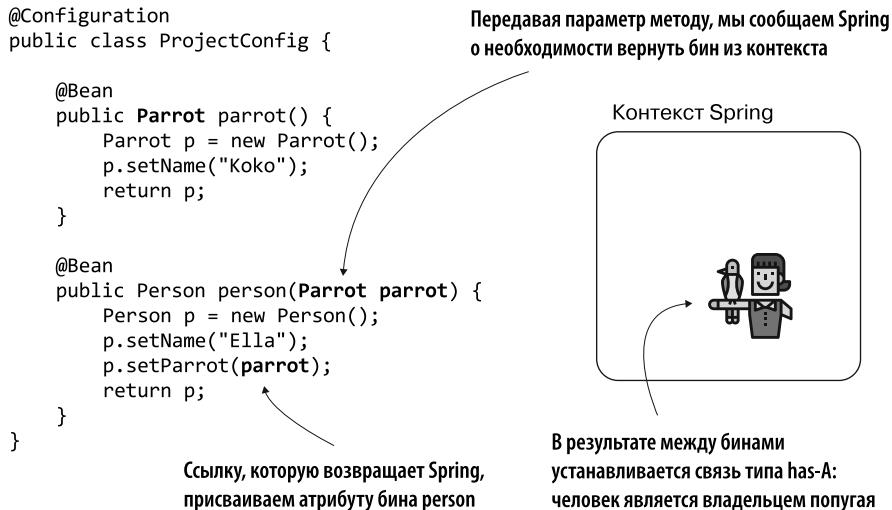


Рис. 3.7. Передавая методу параметр, мы сообщили Spring, что нужно вернуть из контекста бин того типа, к которому переданный параметр принадлежит. Затем мы использовали этот бин (`parrot`) при создании другого бина (`person`). Таким образом мы установили между двумя бинами связь типа has-A

Для демонстрации монтажа бинов, при котором используется параметр вместо прямого вызова метода с `@Bean`, мы воспользуемся кодом, созданным для проекта `sq-ch3-ex2`, и вставим туда ссылку между двумя экземплярами, добавленными в контекст. Я сохранил этот пример в новом проекте `sq-ch3-ex3`.

В листинге 3.4 содержится определение класса конфигурации. Обратите внимание на метод `person()`. Теперь он принимает параметр типа `Parrot` и присваивает ссылку возвращаемому атрибуту экземпляра `person`. При вызове этого метода Spring найдет в контексте бин `parrot` и внедрит его значение в параметр в методе `person()`.

Листинг 3.4. Внедрение зависимостей бина путем передачи параметра в метод

```
@Configuration
public class ProjectConfig {

    @Bean
    public Parrot parrot() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }

    @Bean
    public Person person(Parrot parrot) { ←—— Spring внедряет бин parrot в этот параметр
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot);
        return p;
    }
}
```

В предыдущем абзаце я употребил слово «внедрение». То, что мы сейчас прописали в листинге, отныне будем называть внедрением зависимостей (dependency injection, DI). Как следует из названия, технология DI состоит в том, что фреймворк присваивает значение определенному полю или параметру. В данном случае Spring присваивает значение параметру метода `person()` при вызове этого метода и устанавливает для него зависимость. DI – это применение принципа IoC, а IoC подразумевает, что фреймворк управляет выполнением приложения. Чтобы освежить вашу память, я повторю на рис. 3.8 то, что вы уже видели в главе 1 (см. рис. 1.4), когда мы говорили об IoC.

Вы будете часто применять DI (и не только в Spring), поскольку это очень удобный способ управления экземплярами создаваемых объектов. DI позволяет сократить количество кода, который приходится писать при разработке приложений.

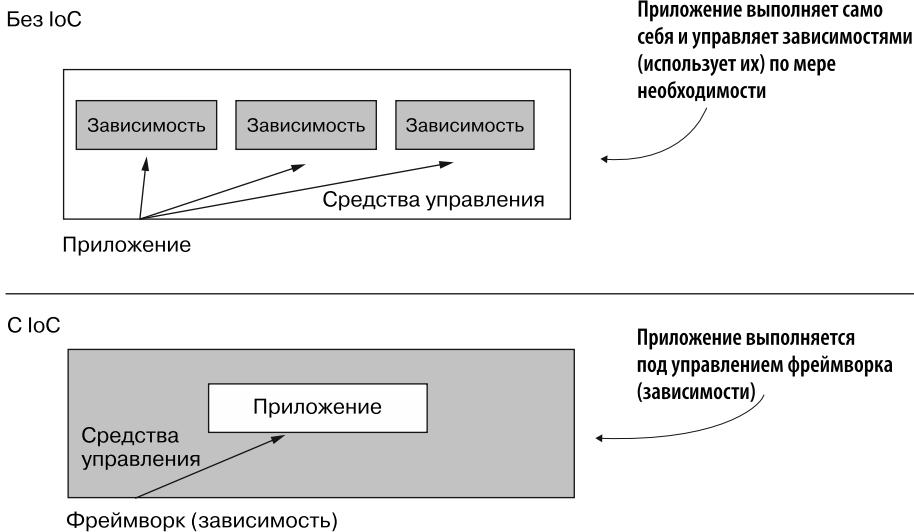


Рис. 3.8. Без использования принципа IoC приложение само контролирует свое выполнение и использует многочисленные зависимости. Приложении, в котором применяется принцип IoC, позволяет зависимости управлять своим выполнением. DI как раз и является примером такого управления. Фреймворк (зависимость) присваивает значение полю объекта, который принадлежит приложению

При выполнении приложения в консоли появятся сообщения, подобные показанным ниже. Вы увидите, что попугай Koko действительно принадлежит человеку Ella:

```
Parrot created
Person's name: Ella
Person's parrot: Parrot : Koko
```

3.2. ВНЕДРЕНИЕ БИНОВ С ПОМОЩЬЮ АННОТАЦИИ @AUTOWIRED

Рассмотрим еще один способ установить связь между бинами, размещенными в контексте Spring. Данная технология, требующая использования аннотации `@Autowired`, будет часто встречаться вам при изменении класса, для которого был определен бин (если этот класс не является частью зависимости). Аннотация `@Autowired` позволяет пометить свойство объекта, в которое мы хотим включить значение из контекста Spring. Мы отмечаем это свойство непосредственно

в классе, характеризующем объект, куда нужно внедрить зависимость. Благодаря такому подходу связь между двумя объектами становится более заметной, чем при использовании других вариантов, описанных в разделе 3.1. Как вы скоро узнаете, есть следующие три способа применения аннотации `@Autowired`:

- внедрение значения в поле класса — именно такие примеры вам будут встречаться в большинстве описаний данной концепции;
- внедрение значения через параметры конструктора класса — этот способ вы будете чаще всего использовать на практике;
- внедрение значения через сеттер — в готовом к эксплуатации коде этот метод встречается очень редко.

Рассмотрим перечисленные варианты более подробно и напишем примеры для каждого из них.

3.2.1. Внедрение значений через поля класса с использованием аннотации `@Autowired`

Начнем с самого простого из трех вариантов использования аннотации `@Autowired`, который также встречается в примерах разработчиков: применение аннотации к полю класса (рис. 3.9). Как вы вскоре узнаете, несмотря на всю простоту, у данного способа есть свои недостатки — именно поэтому мы избегаем его при написании кода для промышленных программных продуктов. Однако он будет часто встречаться вам в примерах, концептуальных прототипах, а также при написании тестов, которые мы рассмотрим в главе 15, так что настоящий прием стоит освоить.

Мы разработаем проект `sq-ch3-ex4`, в котором добавим аннотацию `@Autowired` к полю `parrot` класса `Person`, чтобы Spring внедрил в это поле значение из контекста. Для начала создадим классы, описывающие два объекта, — `Person` и `Parrot`. Описание класса `Parrot` представлено в следующем примере кода:

```
@Component
public class Parrot {

    private String name = "Koko";

    // геттеры и сеттеры

    @Override
    public String toString() {
        return "Parrot : " + name;
    }
}
```

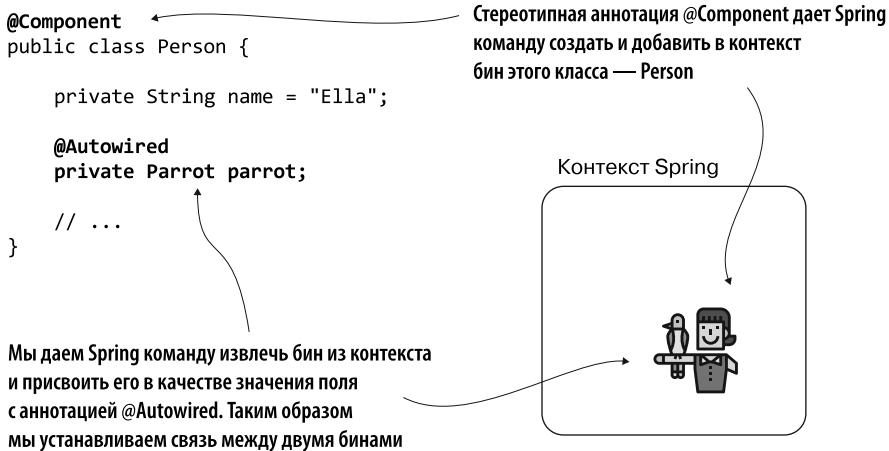


Рис. 3.9. Применяя аннотацию `@Autowired` к полю класса, мы даем Spring команду присвоить этому полю значение бина, полученного из контекста. Spring создает два бина, `person` и `parrot`, и внедряет объект `parrot` в поле бина типа `Person`

Здесь мы воспользовались стереотипной аннотацией `@Component`, с которой познакомились в главе 2 (см. подраздел 2.2.2). Мы использовали ее как альтернативу созданию бина в классе конфигурации. Встретив класс с аннотацией `@Component`, Spring создает экземпляр этого класса и добавляет данный экземпляр в контекст. Определение класса `Person` представлено в следующем фрагменте кода:

```

@Component
public class Person {

    private String name = "Ella";
    @Autowired
    private Parrot parrot;
    // геттеры и сеттеры
}

```

Добавив к полю аннотацию `@Autowired`, мы даем Spring команду внедрить в это поле соответствующее значение из контекста

ПРИМЕЧАНИЕ

В примере выше для добавления бинов в контекст Spring я использовал стереотипные аннотации. Я мог бы определить бины с помощью `@Bean`, но в реальных приложениях вы чаще всего будете встречать аннотацию `@Autowired` в сочетании именно со стереотипными аннотациями, поэтому сейчас мы выберем тот способ, который будет для вас наиболее полезен.

Продолжим наш пример и определим класс конфигурации. Я дал этому классу имя `ProjectConfig` и добавил к нему аннотацию `@ComponentScan`. Как вы уже знаете из главы 2 (см. подраздел 2.2.2), эта аннотация сообщает Spring, где

находятся классы с аннотацией `@Component`. Определение класса конфигурации представлено в следующем фрагменте кода:

```
@Configuration
@ComponentScan(basePackages = "beans")
public class ProjectConfig {

}
```

Затем, как и в предыдущих примерах этой главы, я использовал класс `main`, чтобы убедиться, что Spring правильно внедрил ссылку на `parrot`:

```
public class Main {

    public static void main(String[] args) {
        var context = new AnnotationConfigApplicationContext
            (ProjectConfig.class);

        Person p = context.getBean(Person.class);

        System.out.println("Person's name: " + p.getName());
        System.out.println("Person's parrot: " + p.getParrot());
    }
}
```

В результате в консоль будет выведено примерно следующее. Как следует из второй строки, попугай (в моем случае `Koko`) принадлежит бину, описывающему человека (`Ella`):

```
Person's name: Ella
Person's parrot: Parrot : Koko
```

Почему же этот способ нежелательно применять в рабочем коде? Не то чтобы его там вообще нельзя было использовать, однако в реальных приложениях необходимо гарантировать удобство поддержки и тестирования кода. Внедрение значения непосредственно в поле класса имеет следующие недостатки:

- нет возможности отметить это поле как `final` (см. следующий фрагмент кода) и таким образом гарантировать, что никто не сможет изменить его после того, как оно будет использовано:

```
@Component
public class Person {

    private String name = "Ella";

    @Autowired
    private final Parrot parrot; ← | Этот код не пройдет компиляцию. Поле без начального
                                | значения нельзя определить как final
}
```

- при инициализации приходится самостоятельно управлять значением, а это сложнее.

Как вы узнаете из главы 15, иногда приходится создавать экземпляры объектов, чтобы было проще управлять зависимостями в модульных тестах.

3.2.2. Использование аннотации `@Autowired` для внедрения значения через конструктор

Второй способ внедрения значений в атрибуты объектов при создании бина Spring — это использование конструктора класса, который определяет экземпляр этого класса (рис. 3.10).

Данный способ является наиболее распространенным в промышленном коде, и именно его я рекомендую использовать. Он позволяет объявлять поля как `final`, таким образом гарантируя, что никто не сможет изменить значения этих полей после их инициализации в Spring. Присваивание значений при вызове конструктора также поможет вам при написании специфических модульных тестов, где нельзя полагаться на то, что Spring сам сделает внедрение в поле класса (позже мы подробнее рассмотрим этот вопрос).

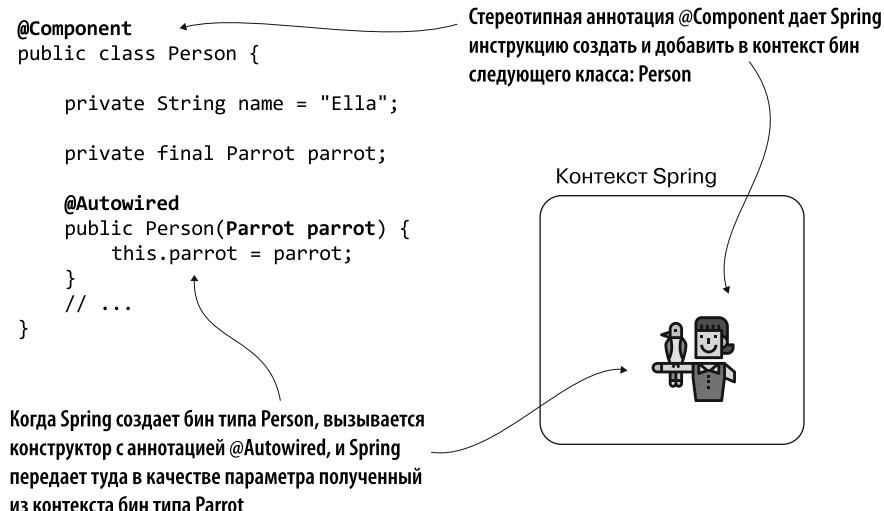


Рис. 3.10. Если определить конструктор с параметром, то при вызове конструктора Spring присвоит параметру бин, полученный из контекста

Мы можем легко переделать реализацию проекта из подраздела 3.2.1 так, чтобы в нем вместо внедрения в поле использовалось внедрение в конструктор. Для этого достаточно изменить класс `Person`, как показано в листинге 3.5. Необходимо определить конструктор класса и снабдить его аннотацией `@Autowired`. Теперь можно установить поле `parrot` как `final`. Никакие изменения в классе конфигурации не нужны.

Листинг 3.5. Внедрение значений через конструктор

```

@Component
public class Person {

    private String name = "Ella";           | Теперь можно пометить это поле
                                              | как final, чтобы его значение
                                              | не изменилось после инициализации

    @Autowired <-----| public Person(Parrot parrot) {
        this.parrot = parrot;                | Мы снабдили конструктор
                                              | аннотацией @Autowired

    }
    // геттеры и сеттеры
}

```

Чтобы сохранить все этапы и изменения, я выделил этот пример в отдельный проект sq-ch3-ex5. Запустив приложение, мы увидим, что в консоль выводятся те же результаты, что и в примере из подраздела 3.2.1. Как видим, человек является владельцем попугая, так что Spring правильно установил связь между экземплярами:

```

Person's name: Ella
Person's parrot: Parrot : Koko

```

ПРИМЕЧАНИЕ

Начиная со Spring 4.3, если класс состоит только из одного конструктора, аннотацию @Autowired можно пропустить.

3.2.3. Внедрение зависимости через сеттер

Разработчики используют внедрение зависимостей в сеттерах нечасто. У этого способа больше недостатков, чем преимуществ: код сложнее читать, нельзя пометить поле как `final`, тестирование не становится проще. Тем не менее считаю важным упомянуть о подобном приеме. Однажды он может вам встретиться, и я бы не хотел, чтобы в тот момент его существование было для вас загадкой. Я не рекомендую его использовать, однако он мне попадался в нескольких старых приложениях. Пример внедрения в сеттере вы найдете в проекте sq-ch3-ex6. Как вы заметите, для этого пришлось лишь изменить класс `Person`. В следующем фрагменте кода я использовал для сеттера аннотацию `@Autowired`:

```

@Component
public class Person {

    private String name = "Ella";

    private Parrot parrot;

    // другие геттеры и сеттеры
}

```

```

@Autowired
public void setParrot(Parrot parrot) {
    this.parrot = parrot;
}
}

```

При запуске приложения вы получите в консоли те же результаты, что и в примерах, рассмотренных в этом разделе ранее.

3.3. ЦИКЛИЧЕСКИЕ ЗАВИСИМОСТИ

Очень удобно, когда созданием и назначением всех зависимостей занимается Spring. Передав фреймворку всю работу, вам не приходится писать множество строк кода, а оставшуюся часть приложения легче читать и понимать. Однако иногда Spring может вносить путаницу. На практике часто встречаются ситуации, когда фреймворк по ошибке создает циклическую зависимость.

Циклическая зависимость (рис. 3.11) — это ситуация, когда для создания бина (назовем его Bean A) Spring должен внедрить зависимость от другого бина, пока еще не существующего (Bean B). Но Bean B, в свою очередь, также требует зависимости от Bean A. И Spring заходит в тупик: он не может создать Bean A, так как для этого нужен Bean B, и не может создать Bean B, так как для этого нужен Bean A.

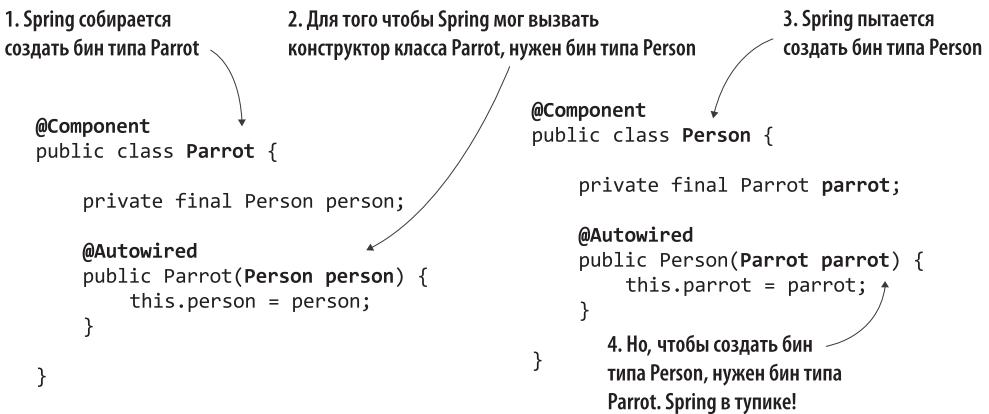


Рис. 3.11. Циклическая зависимость: Spring должен создать бин типа Parrot. Однако, поскольку у Parrot есть зависимость от Person, сначала нужно создать Person. Но, чтобы создать Person, нужно, чтобы уже существовал Parrot. Spring в тупике: он не может создать Parrot, поскольку для этого нужен Person, и не может создать Person, поскольку для этого нужен Parrot

Циклической зависимости легко избежать. Достаточно убедиться, что у вас нет объектов, при создании которых возникает взаимная зависимость этих объектов

98 Часть I. Основные принципы

друг от друга. Наличие такой зависимости — пример плохой разработки классов. В этом случае нужно переписать код.

Я не могу припомнить ни одного Spring-разработчика, который хотя бы раз в жизни не создал циклическую зависимость в своем приложении. Просто помните о такой проблеме — тогда, столкнувшись с ней, вы будете знать ее причины и быстро ее решите.

Пример циклической зависимости вы найдете в проекте sq-ch3-ex7. Как показано в следующем фрагменте кода, я сделал так, чтобы для создания бина `Parrot` требовался бин `Person` и наоборот.

Класс `Person`:

```
@Component
public class Person {

    private final Parrot parrot;

    @Autowired
    public Person(Parrot parrot) { ← Для создания экземпляра Person нужен бин Parrot
        this.parrot = parrot;
    }

    // Остальной код
}
```

Класс `Parrot`:

```
public class Parrot {

    private String name = "Koko";

    private final Person person;

    @Autowired
    public Parrot(Person person) { ← Для создания экземпляра Parrot нужен бин Person
        this.person = person;
    }

    // Остальной код
}
```

При попытке выполнить приложение с подобной конфигурацией получим примерно следующее исключение:

```
Caused by:
org.springframework.beans.factory.BeanCurrentlyInCreationException: Error
creating bean with name 'parrot': Requested bean is currently in creation:
Is there an unresolvable circular reference?
    at
org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.before
SingletonCreation(DefaultSingletonBeanRegistry.java:347)
```

Этим исключением Spring пытается сообщить о возникшей проблеме. Смыл сообщения вполне понятен: фреймворк столкнулся с циклической зависимостью и говорит о том, какие именно классы ее вызвали. Если вам встретится подобное исключение, нужно обратиться к перечисленным в нем классам и убрать циклическую зависимость.

3.4. ВЫБОР ИЗ НЕСКОЛЬКИХ БИНОВ В КОНТЕКСТЕ SPRING

Рассмотрим ситуацию, когда Spring должен внедрить значение в параметр или поле класса, но в контексте существует несколько бинов одного типа и нужно выбрать один из них. Предположим, в контексте Spring есть три бина типа `Parrot`. Согласно конфигурации фреймворк должен включить значение типа `Parrot` в параметр. Что будет делать Spring в этом случае? Какой из нескольких бинов данного типа фреймворк выберет для внедрения?

В зависимости от конкретной реализации возможны следующие варианты.

1. Идентификатор параметра совпадает с именем одного из бинов, добавленных в контекст (которое, напомню, в свою очередь, идентично имени метода, снабженного аннотацией `@Bean` и возвращающего значение этого бина). В подобном случае Spring выберет бин с таким же именем, как и у параметра.
2. Идентификатор параметра не совпадает ни с одним из имен бинов, имеющихся в контексте. Тогда можно поступить следующим образом:
 - отметить один из бинов как первичный (с помощью аннотации `@Primary`, как в главе 2). В этом случае Spring выберет для внедрения первичный бин;
 - выбрать некий бин и отметить его аннотацией `@Qualifier`, как будет описано далее;
 - не делать ничего из выше перечисленного — однако в подобном случае приложение завершится ошибкой и выдаст исключение, сообщающее о том, что в контексте есть несколько бинов одного типа и Spring не может выбрать один из них.

Далее мы будем работать с проектом `sq-ch3-ex8`, где в контексте Spring есть несколько экземпляров одного типа. В листинге 3.6 показан класс конфигурации, в котором определены два экземпляра `Parrot` и используется внедрение через параметры метода.

Запустив приложение с такой конфигурацией, получим в консоли следующий результат. Обратите внимание на то, что Spring связал бин, описывающий человека, с бином попугая `Miki`, так как имя этого бина — `parrot2` (рис. 3.12):

```
Parrot created
Person's name: Ella
Person's parrot: Parrot : Miki
```

Листинг 3.6. Внедрение через параметр в случае нескольких бинов одного типа

```
@Configuration
public class ProjectConfig {
```

```
    @Bean
    public Parrot parrot1() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }
```

```
    @Bean
    public Parrot parrot2() { ← Имя параметра совпадает с именем
        Parrot p = new Parrot();
        p.setName("Miki");
        return p;
    }
```

```
    @Bean
    public Person person(Parrot parrot2) {
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot2);
        return p;
    }
}
```

```
@Configuration
public class ProjectConfig {
```

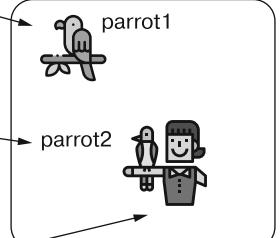
```
    @Bean
    public Parrot parrot1() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }
```

```
    @Bean
    public Parrot parrot2() {
        Parrot p = new Parrot();
        p.setName("Miki");
        return p;
    }
```

```
    @Bean
    public Person person(Parrot parrot2) {
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot2);
        return p;
    }
}
```

В контексте определены два бина типа Parrot.
Имена этих бинов соответствуют названиям
методов, которые их создают: parrot1 и parrot2.
Также определен один бин типа Person

Контекст Spring



Spring выбирает значение того бина,
имя которого совпадает с названием
созданного нами параметра

Рис. 3.12. Один из способов сообщить Spring, какой именно из экземпляров следует выбрать, если в контексте есть несколько экземпляров одного типа, — это выбор по имени. Присвойте параметру то же имя, что и необходимому экземпляру

На практике я стараюсь не опираться на имя параметра, так как эти имена часто меняются при рефакторинге или по ошибке. Для надежности я обычно предпочитаю более наглядный вариант, подчеркивающий мое стремление внедрить определенный бин: аннотацию `@Qualifier`. Но и здесь, как показывает опыт, мнения разделились: многие разработчики выступают как за, так и против этой аннотации. Я считаю, что в данном случае лучше применять именно аннотацию `@Qualifier`, поскольку она явно обозначает ваши намерения. Другие разработчики полагают, что при ее использовании создается ненужный (шаблонный) код.

В листинге 3.7 применяется аннотация `@Qualifier`. Обратите внимание, что вместо того, чтобы создавать специальный идентификатор для параметра, я указал, какой бин нужно внедрить, используя значение атрибута аннотации `@Qualifier`.

Листинг 3.7. Применение аннотации `@Qualifier`

```
@Configuration
public class ProjectConfig {

    @Bean
    public Parrot parrot1() {
        Parrot p = new Parrot();
        p.setName("Koko");
        return p;
    }

    @Bean
    public Parrot parrot2() {
        Parrot p = new Parrot();
        p.setName("Miki");
        return p;
    }

    @Bean
    public Person person(
        @Qualifier("parrot2") Parrot parrot) { ←
        Person p = new Person();
        p.setName("Ella");
        p.setParrot(parrot);
        return p;
    }
}
```

С помощью аннотации `@Qualifier` можно явно выразить свое намерение внедрить определенный бин из контекста

Снова запустив приложение, получим в консоли тот же результат:

```
Parrot created
Person's name: Ella
Person's parrot: Parrot : Miki
```

Аналогичная ситуация получается и при использовании аннотации `@Autowired`. Чтобы это продемонстрировать, я создал еще один проект — `sq-ch3-ex9`. В нем

мы создадим два бина типа `Parrot` (с аннотацией `@Bean`) и экземпляр `Person` (со стереотипной аннотацией). И настроим Spring так, чтобы один из двух бинов типа `Parrot` внедрялся в бин типа `Person`.

Как показано в следующем фрагменте кода, я не добавил в класс `Parrot` аннотацию `@Component`, поскольку собираюсь создать в классе конфигурации два бина типа `Parrot` с помощью аннотации `@Bean`:

```
public class Parrot {  
  
    private String name;  
  
    // геттеры, сеттеры и функция toString()  
}
```

Мы определили бин `Person`, использовав стереотипную аннотацию `@Component`. Обратите внимание на идентификатор, присвоенный параметру конструктора в следующем фрагменте кода. Я назвал этот идентификатор `parrot2`, потому что бин из контекста Spring, который хочу внедрить в этот параметр, назван аналогичным образом:

```
@Component  
public class Person {  
  
    private String name = "Ella";  
  
    private final Parrot parrot;  
  
    public Person(Parrot parrot2) {  
        this.parrot = parrot2;  
    }  
    // геттеры и сеттеры  
}
```

Я создал в классе конфигурации два бина типа `Parrot`, используя аннотацию `@Bean`. Не забудьте: нам еще понадобится аннотация `@ComponentScan`, чтобы сообщить Spring, где находятся классы со стереотипными аннотациями. В данном случае это класс `Person` с аннотацией `@Component`. Определение класса конфигурации показано в листинге 3.8.

Листинг 3.8. Определение бинов типа `Parrot` в классе конфигурации

```
@Configuration  
@ComponentScan(basePackages = "beans")  
public class ProjectConfig {  
  
    @Bean  
    public Parrot parrot1() {  
        Parrot p = new Parrot();  
        p.setName("Koko");  
    }
```

```

        return p;
    }

    @Bean
    public Parrot parrot2() { ← Теперь Spring внедряет бин с именем parrot2 в бин типа Person
        Parrot p = new Parrot();
        p.setName("Miki");
        return p;
    }
}

```

Что произойдет, если выполнить метод `main`, представленный в следующем фрагменте кода? Какой попугай будет принадлежать человеку? Поскольку имя в параметре конструктора совпадает с именем одного из бинов в контексте Spring (`parrot2`), фреймворк внедрит этот бин (рис. 3.13) — как результат, приложение выведет в консоль имя попугая `Miki`:

```

public class Main {

    public static void main(String[] args) {
        var context = new
            AnnotationConfigApplicationContext(ProjectConfig.class);

        Person p = context.getBean(Person.class);

        System.out.println("Person's name: " + p.getName());
        System.out.println("Person's parrot: " + p.getParrot());
    }
}

```

```

@Component
public class Person {

    private String name = "Ella";

    private final Parrot parrot;

    public Person(Parrot parrot2) {
        this.parrot = parrot2;
    }
    // ...
}

```

При создании бина типа `Person` фреймворк Spring должен предоставить значение для параметра конструктора. Если в контексте есть несколько бинов одного типа, то Spring выберет тот, имя которого совпадает с названием параметра конструктора

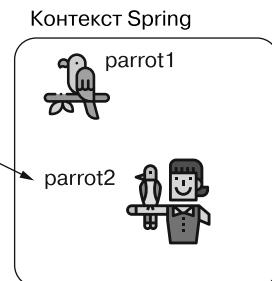


Рис. 3.13. Если в контексте Spring есть несколько бинов одного типа, то Spring выберет тот из них, имя которого совпадает с именем параметра конструктора

Запустив приложение, получим следующий результат:

```
Person's name: Ella
Person's parrot: Parrot : Miki
```

Как и в случае с параметром метода с аннотацией `@Bean`, я не советую полагаться на имя переменной. Вместо этого предлагаю использовать аннотацию `@Qualifier`, чтобы явно показать: я внедряю определенный бин из контекста. Таким образом мы сводим к минимуму вероятность изменения имени переменной при рефакторинге и, как результат, сбоев в работе приложения. В следующем фрагменте кода показаны изменения, которые я внес в класс `Person`. Используя аннотацию `@Qualifier`, я определил имя бина, который должен быть внедрен из контекста Spring, вместо того чтобы полагаться на идентификатор параметра конструктора (эти поправки вы найдете в проекте `sq-ch3-ex10`):

```
@Component
public class Person {

    private String name = "Ella";

    private final Parrot parrot;

    public Person(@Qualifier("parrot2") Parrot parrot) {
        this.parrot = parrot;
    }

    // геттеры и сеттеры
}
```

Поведение приложения не изменится, вывод в консоли останется тем же, но вероятность ошибок в таком коде будет меньше.

РЕЗЮМЕ

- Контекст Spring — это место в памяти приложения, где фреймворк хранит объекты, которыми он должен управлять. Чтобы фреймворк мог управлять объектом, этот объект необходимо добавить в контекст — и у Spring есть ряд средств, позволяющих это сделать.
- При реализации приложения часто приходится ссылаться из одного объекта на другой. Таким образом объект при выполнении своих функций может делегировать некоторые действия. Для реализации этого поведения необходимо установить связи между бинами в контексте Spring.
- Есть три способа установки связей между бинами:
 - прямая ссылка на метод с аннотацией `@Bean` (который создает первый бин) из метода, который создает второй бин. Spring понимает, что вы

ссылается на бин, размещенный в контексте, и, если такой бин уже существует, не вызывает метод создания бина еще раз, а возвращает ссылку на тот, который уже есть;

- определение параметра для метода с аннотацией `@Bean`. Когда Spring встречает метод с аннотацией `@Bean` и у этого метода есть параметр, то фреймворк ищет в контексте бин идентичного с параметром типа и подставляет в качестве значения параметра данный бин;
- использование аннотации `@Autowired` следующими способами:
 - ◆ добавлением аннотации `@Autowired` к полю класса, чтобы Spring внедрил в это поле бин из контекста. Данный вариант часто встречается в примерах и демонстрациях концепции;
 - ◆ добавлением аннотации `@Autowired` к конструктору, который Spring будет вызывать при создании бина. Тогда Spring внедрит другие бины из контекста в виде параметров конструктора. Такой способ чаще всего применяется в реальных приложениях;
 - ◆ добавлением аннотации `@Autowired` к сеттеру атрибута, в который Spring должен внедрить бин из контекста. Этот способ редко используется на практике.
- Всякий раз, когда Spring предоставляет значение или ссылку через атрибут класса, метод или параметр конструктора, это делается посредством DI – технологии, построенной по принципу IoC.
- При создании двух бинов, зависящих друг от друга, возникает циклическая зависимость. Spring не может создать бины с циклической зависимостью, и при выполнении такого приложения возникает исключение. Работая с бинами, убедитесь, что между ними не возникает циклической зависимости.
- Если в контексте Spring есть несколько бинов одного типа, фреймворк не может взять из них один для внедрения самостоятельно. Поэтому нужно указать Spring, какой из экземпляров нужно выбрать:
 - с помощью аннотации `@Primary`, благодаря которой при внедрении зависимости один из бинов выбирается по умолчанию;
 - присвоив бинам имена и внедряя их по именам с помощью аннотации `@Qualifier`.

Контекст Spring: использование абстракций

В этой главе

- ✓ Использование интерфейсов для определения контрактов.
- ✓ Применение абстракций бинов в контексте Spring.
- ✓ Использование внедрения зависимостей для абстракций.

Далее мы поговорим о применении абстракций для бинов Spring. Это исключительно важная тема, поскольку в реальных проектах часто приходится использовать абстракции для разделения реализаций.

Для начала в разделе 4.1 мы освежим знания об использовании интерфейсов для определения контрактов. Чтобы раскрыть эту тему, рассмотрим обязанности объектов и разберемся, как они соответствуют структуре стандартных классов приложения. Используя имеющиеся у вас навыки программирования, мы напишем небольшой сценарий, в котором, пока не используя Spring, сконцентрируемся на реализации требований и применении абстракций для разделения зависимых объектов приложения.

Затем в разделе 4.2 мы рассмотрим поведение Spring в случае применения DI к абстракциям. Мы возьмем за основу проект, созданный в разделе 4.1, и добавим Spring в зависимости приложения. Затем с помощью контекста Spring внедрим зависимость. На этом примере вы ближе познакомитесь с тем, что с большой долей вероятности найдется в готовых приложениях: с объектами, имеющими типичные обязанности в реальных сценариях работы, и с применением абстракций посредством DI и контекста Spring.

4.1. ПРИМЕНЕНИЕ ИНТЕРФЕЙСОВ ДЛЯ ОПРЕДЕЛЕНИЯ КОНТРАКТОВ

Обсудим использование интерфейсов для определения контрактов. Интерфейс Java — это абстрактная структура, которая применяется для объявления определенных обязанностей. Эти обязанности даются объекту, реализующему интерфейс. Другие объекты этого же интерфейса могут выполнять эти обязанности другими способами. Можно сказать, что интерфейс описывает «то, что должно случиться», а каждый отдельный объект — «то, как именно это должно случиться».

Однажды в детстве отец подарил мне старенький радиоприемник, чтобы я его разобрал и играл с деталями. (Мне ужасно нравилось разбирать всякие приборы.) Посмотрев на приемник, я понял, что мне нужен какой-то инструмент, чтобы открутить болты, удерживающие корпус. Слегка поразмыслив, я решил, что нож для этого вполне подойдет, и попросил его у отца. Он спросил: «Зачем тебе нож?» Я ответил, что хочу вскрыть корпус. «А, — сказал он, — тогда тебе лучше взять отвертку. Держи!» Так я узнал, что в любой непонятной ситуации всегда лучше сказать, зачем тебе что-то нужно, чем брать готовое решение самостоятельно. Так и для объектов интерфейс — это способ получить то, что им нужно.

4.1.1. Использование интерфейсов для разделения реализаций

Далее мы поговорим о контрактах и о том, как их создать в Java-приложении посредством интерфейсов. Чтобы раскрыть эту концепцию и показать, в каких случаях стоит использовать интерфейсы, я начну с аналогии, а затем продемонстрирую несколько рисунков. Потом, в подразделе 4.1.2, мы поговорим о требованиях, предъявляемых к задаче, а в подразделе 4.1.3 решим ее без использования фреймворка. Затем в разделе 4.2 мы добавим в наше блюдо Spring — и вы увидите, как внедрение зависимостей в Spring сочетается с использованием контрактов для разделения функционала.

Проведу параллель: предположим, что вам нужно куда-то поехать и вы ищете попутчиков с помощью специального приложения. Скорее всего, вас не интересует внешний вид машины или личность водителя. Вам просто нужно доехать. Мне, к примеру, совершенно безразлично, кто подберет меня на дороге — автомобиль или космический корабль, — если он вовремя доставит меня куда следует. Приложение поиска попутчиков — это и есть интерфейс. Пассажир заказывает не машину или водителя, а поездку. На запрос пассажира может отозваться любой водитель, способный выполнить поставленную задачу. Пассажир и водитель разделены приложением (интерфейсом). Пока машина не найдена, пассажир не знает, ни кто является водителем, ни какая машина его подберет. Водителю тоже неинтересно, кому он оказывает услугу. Используя эту

параллель, мы можем сделать вывод о роли интерфейсов во взаимоотношениях между объектами в Java.

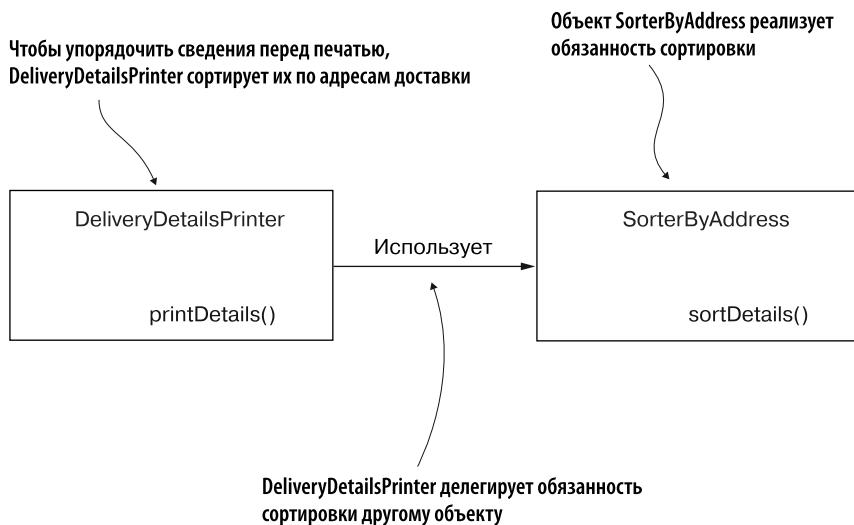


Рис. 4.1. Объект `DeliveryDetailsPrinter` делегирует обязанность сортировки сведений о доставке по адресам доставки другому объекту — `SorterByAddress`

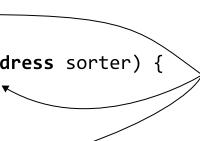
Вот пример реализации: предположим, вы создали объект, который должен печатать сведения о посылках, которыми занимается приложение доставки. Эти сведения должны быть упорядочены по адресу назначения. Объект, осуществляющий их печать, должен делегировать обязанность сортировки посылок по адресам доставки другому объекту (рис. 4.1).

Как показано на рис. 4.1, `DeliveryDetailsPrinter` передает задачу сортировки непосредственно объекту `SorterByAddress`. Если оставить структуру классов как есть, то впоследствии, если понадобится изменить функционал, могут возникнуть сложности. Предположим, что позже придется поменять порядок сортировки информации, чтобы она была упорядочена по имени отправителя. Для этого нужно будет заменить объект `SorterByAddress` на другой, реализующий по-новому сформулированную обязанность. Но изменения также коснутся и объекта `DeliveryDetailsPrinter`, который также затронет выполнение задачи (рис. 4.2).

Как улучшить эту структуру? В случае изменения обязанностей одного объекта не хотелось бы затрагивать и другие объекты. Такая структурная проблема возникает потому, что объект `DeliveryDetailsPrinter` определяет и то, что ему надо, и то, как именно ему это надо. Как уже говорилось, объект должен только сообщить, что ему нужно, но его не должно интересовать, как именно это будет реализовано. Разумеется, мы выполним задачу с помощью интерфейсов. На рис. 4.3

я изобразил интерфейс `Sorter`, который разделяет данные два объекта. Вместо того чтобы декларировать `SorterByAddress`, объект `DeliveryDetailsPrinter` только сообщает, что ему нужен `Sorter`. Теперь мы можем создать сколько угодно объектов, делающих то, что нужно `DeliveryDetailsPrinter`. Любой объект, реализующий интерфейс `Sorter`, может в любой момент удовлетворить зависимость объекта `DeliveryDetailsPrinter`.

```
public class DeliveryDetailsPrinter {
    private SorterByAddress sorter; ←
    public DeliveryDetailsPrinter(SorterByAddress sorter) { ←
        this.sorter = sorter; ←
    }
    public void printDetails() { ←
        sorter.sortDetails(); ←
        // печать сведений о доставке
    }
}
```



При необходимости изменить обязанность сортировки придется модифицировать код в этих местах

Рис. 4.2. Поскольку эти два объекта крепко связаны, то при необходимости внести корректизы в обязанность сортировки приходится менять и тот объект, который участвует в выполнении задачи. Более совершенная структура позволяла бы модифицировать только обязанность сортировки, не меняя объект, задействованный в процессе

На рис. 4.3 представлена наглядная иллюстрация зависимости между объектами `DeliveryDetailsPrinter` и `SorterByAddress` после того, как они были разделены посредством интерфейса.

Определение интерфейса `Sorter` представлено в следующем фрагменте кода:

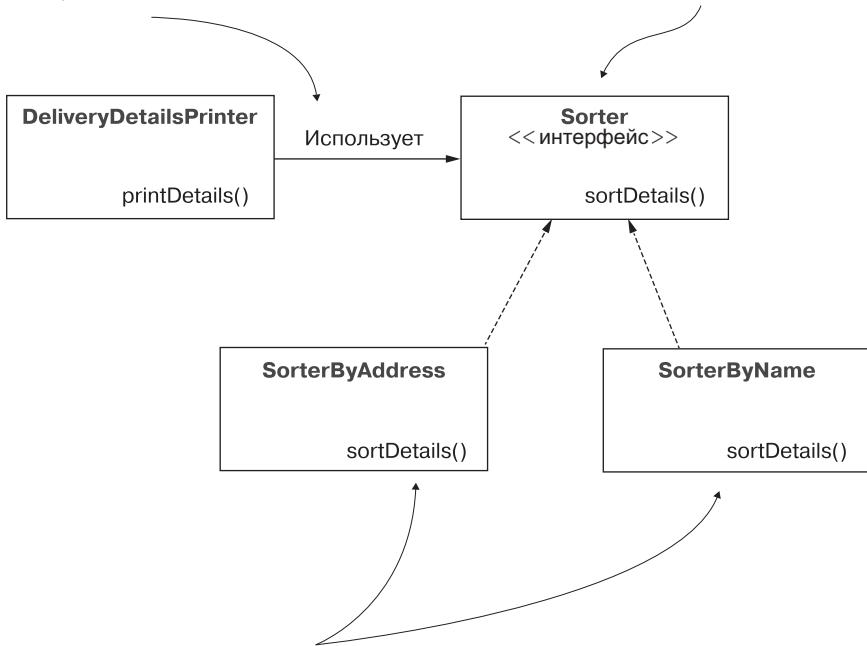
```
public interface Sorter {
    void sortDetails();
}
```

Посмотрите на рис. 4.4 и сравните его с рис. 4.2. Поскольку объект `DeliveryDetailsPrinter` зависит не от реализации, а от интерфейса, нам больше не нужно изменять этот объект, если мы захотим модифицировать способ сортировки сведений о доставке.

После теоретического вступления вы понимаете, зачем нужны интерфейсы для разделения объектов, зависящих друг от друга согласно структуре классов. Теперь мы реализуем требование нашей задачи. Мы построим решение на чистом Java, без фреймворка, и обратим внимание на обязанности объектов и на разделение объектов посредством интерфейсов. В конце раздела мы создадим проект, в котором определим несколько объектов, взаимодействующих между собой для реализации данного сценария.

В объекте DeliveryDetailsPrinter указано только то, что нужно для реализации обязанности объекта. Теперь объект DeliveryDetailsPrinter зависит не от реализации, а от интерфейса

Интерфейс Sorter определяет, что нужно объекту DeliveryDetailsPrinter



Теперь можно создать несколько объектов, реализующих один интерфейс. Так мы сможем изменять реализацию («то, как именно это должно случиться»), не затрагивая объект, который получает результаты от данной реализации (DeliveryDetailsPrinter)

Рис. 4.3. Применение интерфейса для разделения обязанностей. Теперь объект DeliveryDetailsPrinter зависит не от самой реализации, а от интерфейса (контракта). DeliveryDetailsPrinter больше не привязан к определенной реализации и может использовать любой объект, реализующий интерфейс

```

public class DeliveryDetailsPrinter {

    private Sorter sorter;

    public DeliveryDetailsPrinter(Sorter sorter) {
        this.sorter = sorter;
    }

    public void printDetails() {
        sorter.sortDetails();
        // печать сведений о доставке
    }
}
  
```

Теперь можно применять любую реализацию интерфейса Sorter, и не потребуется изменять объект, использующий эту обязанность

Рис. 4.4. Объект DeliveryDetailsPrinter зависит от интерфейса Sorter. Можно изменить реализацию интерфейса Sorter, не меняя объект, использующий эту обязанность (DeliveryDetailsPrinter)

В разделе 4.2 мы изменим проект, добавив в него Spring, который будет управлять объектами и связями между ними посредством внедрения зависимостей. Благодаря такому поэтапному подходу вам будет легче заметить, что необходимо изменить в коде, чтобы подключить Spring к приложению, а также какие преимущества такое подключение дает.

4.1.2. Условия задачи

До сих пор мы рассматривали несложные примеры и использовали простые объекты (такие как `Parrrot`). Они весьма далеки от реальных промышленных приложений, зато помогают сконцентрироваться на изучаемых в данный момент синтаксических конструкциях. Пора пойти дальше и использовать то, что мы узнали в предыдущих главах, на примере, более близком к реальным условиям.

Предположим, нам нужно создать приложение, с помощью которого команда работников управляет своими обязанностями. Одна из функций продукта — возможность оставлять заметки к задачам. Когда пользователи публикуют комментарий, он где-то сохраняется (например, в базе данных) — и приложение отправляет письмо по электронному адресу, указанному в конфигурации приложения.

Необходимо разработать объекты, правильно распределить между ними обязанности и создать абстракции для реализации этой функции.

4.1.3. Реализация сценариев использования без применения фреймворка

Сконцентрируемся на решении задачи, описанной в подразделе 4.1.1, используя то, что мы уже знаем об интерфейсах. Прежде всего нам нужно определить объекты (и обязанности), которые будут реализованы.

На практике в типичных приложениях объекты, выполняющие условия задачи, обычно называют сервисами — и мы тоже воспользуемся данным термином. Нам нужен сервис, который реализует сценарий использования «напечатать комментарий». Назовем этот сервис `CommentService`. Я предпочитаю давать классам сервисов имена, заканчивающиеся на `service`, чтобы подчеркнуть их роль в проекте. (Подробнее о рекомендациях по выбору имен советую почитать главу 2 книги Роберта Мартина «Чистый код. Создание, анализ и рефакторинг» (Питер, 2021).)

Снова проанализировав требования системы, мы обнаружим, что данный сценарий использования состоит из двух частей: сохранения комментария и его отправки по электронной почте. Поскольку это совершенно разные действия, лучше разделить их на две обязанности, соответственно, нам понадобится реализовать два разных объекта.

Объект, который непосредственно взаимодействует с базой данных, обычно называют *репозиторием*. Встречается также термин «*объект доступа к данным*» (data access object, DAO). Поэтому первому объекту, реализующему обязанность сохранения комментария, мы дадим имя *CommentRepository*.

В реальных приложениях объекты, коммуницирующие с чем-то, что находится за пределами приложения, принято называть прокси. Поэтому второй объект, обязанностью которого является отправка электронных писем, мы назовем *CommentNotificationProxy*. Связи между нашими объектами показаны на рис. 4.5.



Рис. 4.5. Объект CommentService выполняет сценарий использования «опубликовать комментарий». Для этого он делегирует обязанности, реализованные в объектах CommentRepository и CommentNotificationProxy

Но постойте! Ведь только недавно было сказано, что не следует допускать прямых связей между реализаций! Необходимо разделить эти реализации посредством интерфейсов. Например, сейчас *CommentRepository*, очевидно, будет сохранять комментарии в базе данных. Но в будущем, возможно, его понадобится заменить на какую-нибудь другую технологию или внешний сервис. То же самое можно сказать и о *CommentNotificationProxy*. Сейчас он отправляет сообщения по электронной почте, но в следующих версиях уведомления могут пойти по какому-нибудь другому каналу. Нам обязательно нужно разделить *CommentService* и реализации его зависимостей, чтобы затем, когда эти зависимости потребуется изменить, не пришлось затронуть и объект, их использующий.

На рис. 4.6 показано, как разделить структуру этого класса посредством абстракций. Мы сделаем *CommentRepository* и *CommentNotificationProxy* не классами, а интерфейсами, которые затем реализуем, чтобы определить их функционал.

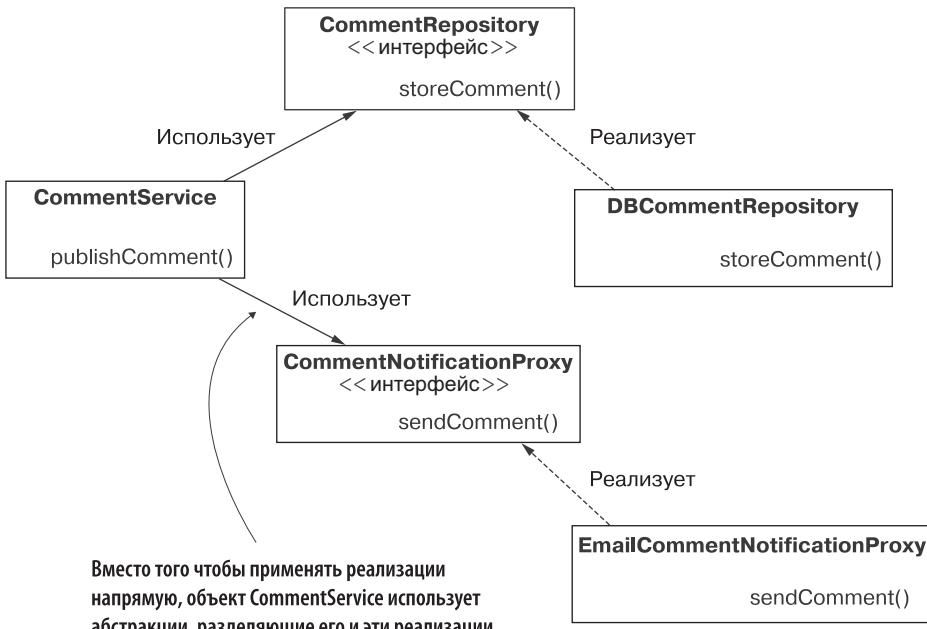


Рис. 4.6. Объект CommentService зависит от абстракций, представленных интерфейсами CommentRepository и CommentNotificationProxy. Эти интерфейсы, в свою очередь, реализуются классами DBCommentRepository и EmailCommentNotificationProxy. В такой структуре выполнение сценария «опубликовать комментарий» отделено от зависимостей, благодаря чему будет проще изменять приложение в ходе дальнейшей разработки

Теперь, получив ясное представление о том, что мы хотим построить, начнем писать код. Для начала создадим обычный проект Maven, без каких-либо внешних зависимостей в файле `pom.xml`. Я назову этот проект `sq-ch4-ex1` и построю его так, как показано на рис. 4.7, с обязанностями, размещенными в отдельных пакетах.

И еще один момент, о котором я пока не упоминал, чтобы вы сфокусировались на главных обязанностях приложения: комментарии нужно как-то представлять. Для этого вначале нам нужно написать небольшой класс POJO. Его обязанность — моделировать данные, которые использует приложение. Такие объекты называют *моделями*. Будем считать, что у комментария есть два атрибута: текст и автор. Мы создадим пакет `model1`, где опишем класс `Comment`. Определение этого класса представлено в листинге 4.1.

ПРИМЕЧАНИЕ

POJO — это простой объект без зависимостей, описываемый только своими атрибутами и методами. В нашем случае класс `Comment` определяет объект POJO, характеризуя свойства комментария с помощью двух атрибутов: `author` и `text`.

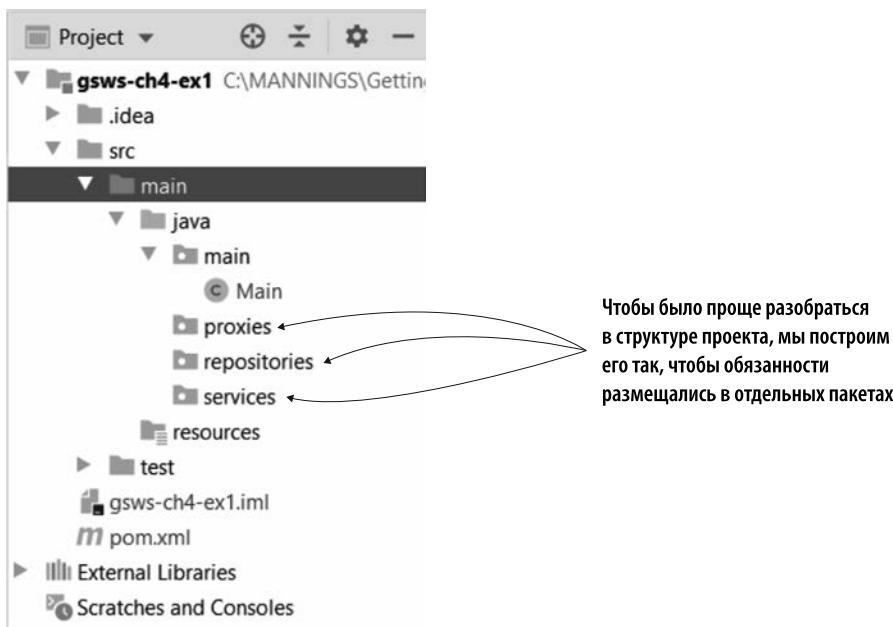


Рис. 4.7. Структура проекта. Мы создали отдельные пакеты для каждой обязанности, чтобы вся конструкция была более понятной и удобной для чтения

Листинг 4.1. Определение комментария

```
public class Comment {  
  
    private String author;  
  
    private String text;  
  
    // геттеры и сеттеры  
}
```

Теперь мы можем назначить обязанности для репозитория и прокси. В листинге 4.2 представлено определение интерфейса `CommentRepository`. Контракт, описываемый этим интерфейсом, объявляет метод `storeComment(Comment comment)`, который необходим объекту `CommentService` для реализации сценария использования. Этот интерфейс и реализующий его класс мы сохраним в пакете `repositories`.

Листинг 4.2. Определение интерфейса `CommentRepository`

```
public interface CommentRepository {  
  
    void storeComment(Comment comment);  
}
```

Данный интерфейс только сообщает, что нужно объекту `CommentService` для реализации сценария использования: сохранение комментария. Когда мы назначим объект, который будет реализовывать этот контракт, он переопределит метод `storeComment(Comment comment)`, чтобы описать, как именно будут сохраняться комментарии. В листинге 4.3 вы увидите определение класса `DBCommentRepository`. Мы еще не знаем, как устанавливать соединение с базой данных (изучим в главе 12), поэтому будем просто выводить текст в консоль, чтобы имитировать это действие.

Листинг 4.3. Реализация интерфейса `CommentRepository`

```
public class DBCommentRepository implements CommentRepository {
    @Override
    public void storeComment(Comment comment) {
        System.out.println("Storing comment: " + comment.getText());
    }
}
```

Аналогичным образом определим интерфейс для второй обязанности объекта `CommentService` — `CommentNotificationProxy`. Этот интерфейс и класс, который его реализует, мы выделим в пакет `proxies`. Интерфейс `CommentNotificationProxy` представлен в листинге 4.4.

Листинг 4.4. Определение интерфейса `CommentNotificationProxy`

```
public interface CommentNotificationProxy {
    void sendComment(Comment comment);
}
```

В листинге 4.5 содержится реализация данного интерфейса, которую мы будем использовать в нашем примере.

Листинг 4.5. Реализация интерфейса `CommentNotificationProxy`

```
public class EmailCommentNotificationProxy
    implements CommentNotificationProxy {
    @Override
    public void sendComment(Comment comment) {
        System.out.println("Sending notification for comment: "
            + comment.getText());
    }
}
```

Теперь можно реализовать сам объект `CommentService` с его двумя зависимостями (`CommentRepository` и `CommentNotificationProxy`). Класс `CommentService`, представленный в листинге 4.6, находится в пакете `service`.

Листинг 4.6. Реализация объекта CommentService

```
public class CommentService {
    private final CommentRepository commentRepository;
    private final CommentNotificationProxy commentNotificationProxy;

    public CommentService(
        CommentRepository commentRepository,
        CommentNotificationProxy commentNotificationProxy) {
        this.commentRepository = commentRepository;
        this.commentNotificationProxy = commentNotificationProxy;
    }

    public void publishComment(Comment comment) {
        commentRepository.storeComment(comment);
        commentNotificationProxy.sendComment(comment);
    }
}
```

Определяем две зависимости в виде атрибутов класса

Представляем эти зависимости в момент создания объекта посредством параметров конструктора

Реализуем сценарий использования, который делегирует зависимостям обязанности «сохранить комментарий» и «отправить уведомление»

Теперь можно написать класс `Main`, представленный в листинге 4.7, и протестировать всю структуру классов.

Листинг 4.7. Вызов сценария использования из класса Main

```
public class Main {

    public static void main(String[] args) {
        var commentRepository =
            new DBCommentRepository();
        var commentNotificationProxy =
            new EmailCommentNotificationProxy();
        var commentService =
            new CommentService(
                commentRepository, commentNotificationProxy);

        var comment = new Comment();
        comment.setAuthor("Laurentiu");
        comment.setText("Demo comment");
        commentService.publishComment(comment);
    }
}
```

Создаем экземпляры для зависимостей

Создаем экземпляр класса сервиса и предоставляем ему зависимости

Создаем экземпляр комментария, чтобы передать его сценарию использования «опубликовать комментарий» в качестве параметра

Вызываем сценарий использования «опубликовать комментарий»

Запустив это приложение, увидим в консоли следующие две строки — они выводятся объектами `CommentRepository` и `CommentNotificationProxy`:

```
Storing comment: Demo comment
Sending notification for comment: Demo comment
```

4.2. ИСПОЛЬЗОВАНИЕ ВНЕДРЕНИЯ ЗАВИСИМОСТЕЙ ДЛЯ АБСТРАКЦИЙ

Далее к структуре классов, построенной в разделе 4.1, мы применим фреймворк Spring. На этом примере мы рассмотрим, как Spring осуществляет внедрение зависимостей при использовании абстракций. Данная тема очень важна, поскольку в большинстве проектов вы будете внедрять зависимости между объектами именно таким образом. В главе 3, чтобы внедрить зависимости, мы использовали конкретные классы для объявления переменных, которым Spring присваивал значения бинов из контекста. Однако, как вы узнаете позднее, Spring также поддерживает и абстракции.

Прежде всего мы добавим в проект зависимость Spring, а затем решим, какие объекты приложения должны находиться под контролем фреймворка. Вы научитесь определять, какие объекты следует отдать под управление Spring, а какие — нет.

Затем мы адаптируем проект, построенный в разделе 4.1, для Spring, с учетом возможностей внедрения зависимостей, предоставляемых фреймворком. Мы обратим особое внимание на различные ситуации, которые возникают при внедрении зависимостей с использованием абстракций. В конце раздела мы рассмотрим еще несколько стереотипных аннотаций. Вы узнаете, что `@Component` не единственная доступная из них и что в разных случаях следует использовать разные аннотации.

4.2.1. Выбор объектов для добавления в контекст Spring

Говоря о Spring в главах 2 и 3, мы обращали внимание на синтаксис, но не на сценарии использования с задачами, возникающими в реальных условиях. В частности, мы не затрагивали вопрос о том, в каких случаях нужно добавлять объект в контекст Spring. Возможно, у вас сложилось ложное впечатление, что в контекст Spring нужно включать все объекты, которые есть в приложении.

Напомню: мы добавляем объекты в контекст Spring для того, чтобы дать фреймворку контроль над ними и расширить их возможности за счет функционала Spring. Поэтому, принимая решение, необходимо ответить на вопрос: «Должен ли фреймворк управлять этим объектом?»

В нашем случае дать ответ нетрудно, поскольку мы используем только одну функцию Spring — DI. Объект, нужный нам для добавления в контекст Spring, либо имеет зависимость, которую мы хотим внедрить из контекста, либо сам

является такой зависимостью. Посмотрев на нашу реализацию, вы заметите, что единственный объект, не соответствующий данному критерию, — это `Comment`. Остальные объекты в нашей структуре классов так или иначе должны попасть под управление Spring:

- `CommentService` — имеет две зависимости, от `CommentRepository` и `CommentNotificationProxy`;
- `DBCommentRepository` — реализует интерфейс `CommentRepository` и является зависимостью для `CommentService`;
- `EmailCommentNotificationProxy` — реализует интерфейс `CommentNotificationProxy` и представляет собой зависимость для `CommentService`.

Но почему бы не добавить в контекст Spring еще и экземпляры `Comment`? Этот вопрос мне часто задают на курсах Spring. Включая в контекст объекты, которые не нуждаются в управлении со стороны фреймворка, мы загромождаем приложение, из-за чего оно становится медленнее и его сложно поддерживать. Специфический функционал фреймворка позволяет сделать управление объектами эффективнее. Но если объект не получит от этого никакой пользы, мы просто переусложним реализацию проекта. В главе 2 говорилось, что самым удобным способом добавления бинов в контекст Spring в случае, если классы принадлежат проекту и вы можете их изменить, является применение стереотипных аннотаций (`@Component`). Мы тоже будем использовать данный способ.

Обратите внимание, что два интерфейса, показанные на рис. 4.8, обозначены белыми квадратами (мы не снабдили их аннотацией `@Component`). Мои студенты часто не понимают, в каких случаях следует использовать стереотипные аннотации, если в приложении есть интерфейсы. Стереотипные аннотации стоит применять для тех классов, для которых Spring создает экземпляры и добавляет эти экземпляры в контекст. У интерфейсов и абстрактных классов не может быть экземпляров, поэтому использовать стереотипные аннотации не имеет смысла. Это возможно с точки зрения синтаксиса, но бесполезно.

Внесем исправления в код и добавим к этим классам аннотацию `@Component`. В листинге 4.8 изменения затронули класс `DBCommentRepository`.

Листинг 4.8. Добавление аннотации `@Component` к классу `DBCommentRepository`

```
@Component ←
public class DBCommentRepository implements CommentRepository {
    @Override
    public void storeComment(Comment comment) {
        System.out.println("Storing comment: " + comment.getText());
    }
}
```

Для класса с аннотацией `@Component` Spring создаст
экземпляр и добавит этот экземпляр в контекст как бин

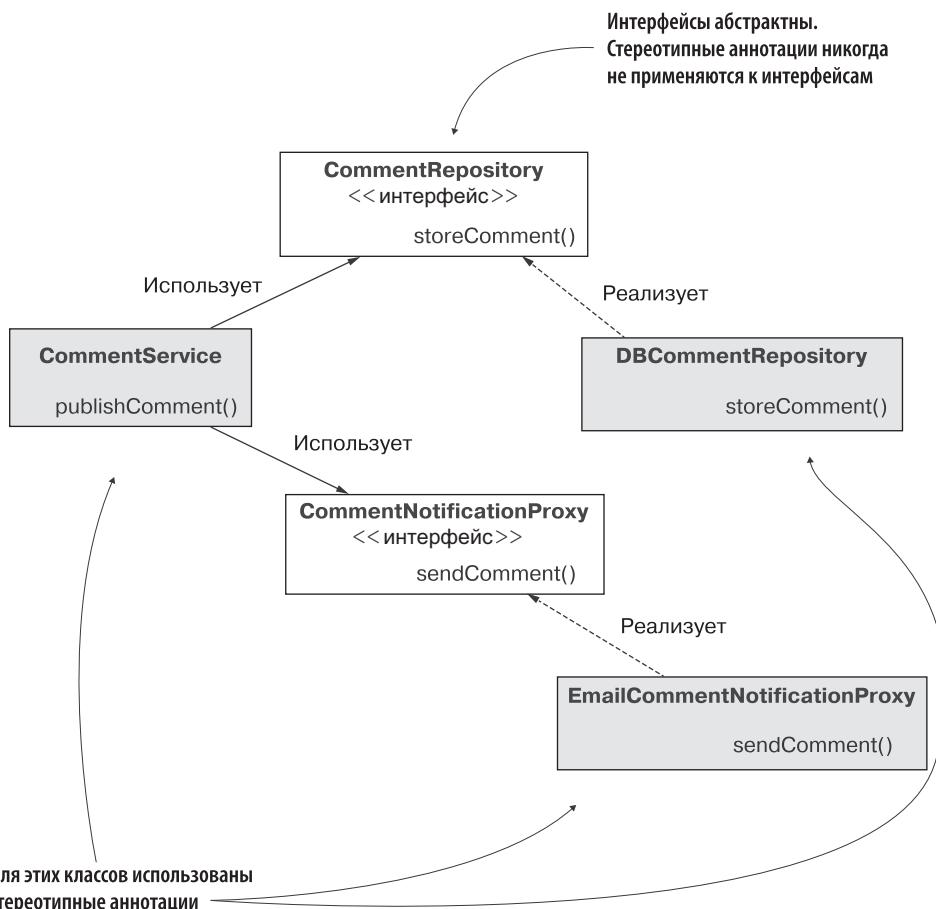


Рис. 4.8. Классы со стереотипными аннотациями @Component выделены серым цветом. При загрузке контекста Spring создаст экземпляры этих классов и добавит их в контекст

В следующем листинге показаны изменения в классе EmailCommentNotificationProxy.

Листинг 4.9. Класс EmailCommentNotificationProxy с аннотацией @Component

```

@Component
public class EmailCommentNotificationProxy
    implements CommentNotificationProxy {

    @Override
    public void sendComment(Comment comment) {
        System.out.println(
  
```

```

        "Sending notification for comment: " +
        comment.getText());
    }
}

```

В листинге 4.10 показан измененный класс `CommentService`, также снабженный аннотацией `@Component`. В классе `CommentService` объявлены зависимости от других двух компонентов, через интерфейсы `CommentRepository` и `CommentNotificationProxy`. Spring — достаточно «умный» фреймворк: встретив атрибуты, определенные посредством интерфейсов, он ищет в контексте бины, принадлежащие классам, которые эти интерфейсы реализуют. Как было показано в главе 2, если в классе есть только один конструктор, то аннотация `@Autowired` не обязательна.

Листинг 4.10. Создание компонента из класса CommentService

```

@Component ← Spring создает бин этого класса и добавляет его в контекст
public class CommentService {

    private final CommentRepository commentRepository;

    private final CommentNotificationProxy commentNotificationProxy;
    | Если бы у этого класса было несколько конструкторов,
    | нужно было бы использовать аннотацию @Autowired

    public CommentService(←
        CommentRepository commentRepository,
        CommentNotificationProxy commentNotificationProxy) {
        this.commentRepository = commentRepository;
        this.commentNotificationProxy = commentNotificationProxy;
    }

    public void publishComment(Comment comment) {
        commentRepository.storeComment(comment);
        commentNotificationProxy.sendComment(comment);
    }
}

```

Spring использует конструктор
для создания бина и при создании
экземпляра внедряет в параметры
ссылки из контекста

Нам осталось только сообщить Spring, где находятся классы со стереотипными аннотациями, и протестировать приложение. В листинге 4.11 представлен класс конфигурации проекта с аннотацией `@ComponentScan`, в которой сообщается, где находятся классы с аннотациями `@Component`. Аннотация `@ComponentScan` была описана в главе 2.

Листинг 4.11. Использование аннотации `@ComponentScan` в классе конфигурации

```

@Configuration ← Класс конфигурации отмечен аннотацией @Configuration
@ComponentScan(←
    basePackages = {"proxies", "services", "repositories"})
)
public class ProjectConfiguration {           | С помощью аннотации @ComponentScan мы сообщаем
                                            | Spring, в каких пакетах находятся классы со стереотипными
                                            | аннотациями. Обратите внимание: пакет model здесь
                                            | не указан, поскольку в нем упомянутых классов нет
}

```

ПРИМЕЧАНИЕ

В этом примере я использовал аннотацию `@ComponentScan` с атрибутом `basePackages`. С его помощью Spring также позволяет указывать классы непосредственно. Преимущество пакетов состоит в том, что нужно указывать только их имена. Даже если в пакете содержится 20 классов компонентов, вам достаточно написать только одно слово (имя пакета), а не все 20. Однако, если разработчик переименует пакет, он может забыть изменить также значение в аннотации `@ComponentScan`. Указывая классы непосредственно, придется писать больше кода, но если кто-нибудь изменит код, то он сразу увидит, что нужно также исправить аннотацию `@ComponentScan`, иначе приложение не будет скомпилировано. В реальных приложениях встречаются оба подхода, и, как показывает мой опыт, ни один из них не лучше другого.

Чтобы протестировать приложение, мы создадим еще один метод, показанный в листинге 4.12. Мы сформируем контекст Spring, извлечем оттуда бин типа `CommentService` и вызовем метод `publishComment(Comment comment)`.

Листинг 4.12. Класс Main

```
public class Main {

    public static void main(String[] args) {
        var context =
            new AnnotationConfigApplicationContext(
                ProjectConfiguration.class);

        var comment = new Comment();
        comment.setAuthor("Laurentiu");
        comment.setText("Demo comment");

        var commentService = context.getBean(CommentService.class);
        commentService.publishComment(comment);
    }
}
```

Запустив приложение, вы увидите в консоли примерно такой результат, подтверждающий, что обе зависимости доступны и правильно вызываются из объекта `CommentService`:

```
Storing comment: Demo comment
Sending notification for comment: Demo comment
```

Это очень маленький пример, и может показаться, что подключение Spring мало что дает. Но посмотрим на него внимательнее: благодаря использованию DI нам не пришлось самим создавать экземпляр объекта `CommentService` со всеми его зависимостями и явно указывать связи между объектами. В реальном приложении, состоящем больше чем из трех классов, разница еще заметнее: возможность передать Spring управление объектами и зависимостями значительно упрощает работу. Применение фреймворка уменьшает количество однотипного кода (который разработчики называют *шаблонным*), что позволяет

сконцентрировать внимание на реальных задачах приложения. Напомню также, что добавление этих экземпляров в контекст дает Spring возможность управлять ими и подключать к ним функции, которые будут описаны в следующих главах.

ВАРИАНТЫ ВНЕДРЕНИЯ ЗАВИСИМОСТЕЙ В СОЧЕТАНИИ С АБСТРАКЦИЯМИ

В главе 3 мы познакомились с несколькими способами применения автомонтажа. Мы изучили аннотацию `@Autowired`, с помощью которой можно сделать внедрение в поле, конструктор или сеттер. Мы также рассмотрели использование автомонтажа в классе конфигурации с применением параметров методов, снабженных аннотацией `@Bean` (эти методы Spring использует для создания бинов в контексте).

Разумеется, в этом разделе я начал с примеров, демонстрирующих подход, который наиболее широко применяется в реальных приложениях, — внедрение через конструктор. Но я считаю, что очень важно знать о существовании других приемов, которые также могут вам встретиться. В данной врезке я хочу подчеркнуть, что DI с абстракциями (как вы уже видели выше) работает точно так же, как и в иных случаях, описанных в главе 3. Чтобы в этом убедиться, попробуем изменить проект `sq-ch4-ex2` так, чтобы в нем вначале использовалось внедрение в поле с помощью аннотации `@Autowired`. Затем мы снова модифицируем проект и проверим, как работает DI с абстракциями при использовании методов с `@Bean` в классе конфигурации.

Чтобы сохранить все этапы нашей работы, я создам для первого примера отдельный проект `sq-ch4-ex3`. К счастью, нам нужно изменить только класс `CommentService`. Мы удалим конструктор и добавим ко всем полям класса аннотацию `@Autowired`, как показано в следующем фрагменте кода:

```
@Component
public class CommentService {
    private CommentRepository commentRepository;
    private CommentNotificationProxy commentNotificationProxy;

    public void publishComment(Comment comment) {
        commentRepository.storeComment(comment);
        commentNotificationProxy.sendComment(comment);
    }
}
```

Эти поля больше не являются `final`, и к ним добавлены аннотации `@Autowired`. Используя конструктор по умолчанию, Spring создает экземпляр этого класса и затем внедряет в него две зависимости из контекста

Как вы, вероятно, ожидаете, теперь через эти параметры можно использовать автомонтаж для методов с `@Bean`, в том числе с абстракциями. Я выделил данные примеры в проект `sq-ch4-ex4`. В нем я полностью удалил из класса `CommentService` стереотипные аннотации (`@Component`) и две их зависимости.

Затем я изменил класс конфигурации, чтобы создать бины и установить связь между ними. Теперь класс конфигурации выглядит так:

```
@Configuration
public class ProjectConfiguration {
    @Bean
    public CommentRepository commentRepository() {
        return new DBCommentRepository();
    }

    @Bean
    public CommentNotificationProxy commentNotificationProxy() {
        return new EmailCommentNotificationProxy();
    }

    @Bean
    public CommentService commentService(
        CommentRepository commentRepository,
        CommentNotificationProxy commentNotificationProxy) {
        return new CommentService(commentRepository,
            commentNotificationProxy);
    }
}
```

Поскольку стереотипные аннотации больше не используются, аннотация `@ComponentScan` тоже не нужна

Создаем бин для каждой из двух зависимостей

С помощью параметров метода `@Bean` (которые теперь имеют тип интерфейса) сообщаем Spring, что нужно предоставить ссылки на бины из контекста; тип этих бинов соответствует типу параметров

4.2.2. Выбор одной из реализаций абстракции для автомонтажа

До сих пор мы изучали поведение Spring при использовании DI с абстракциями. Но мы рассматривали пример, в котором заведомо существовал только один экземпляр каждой внедряемой нами абстракции.

Двигаемся дальше: что получится, если в контексте Spring есть несколько экземпляров, соответствующих запрошенной абстракции? Такое вполне может произойти в реальном приложении, и вам следует знать, что делать в этих случаях, чтобы приложение работало так, как ожидается.

Предположим, у нас есть два бина, принадлежащие двум разным классам, которые реализуют интерфейс `CommentNotificationProxy` (рис. 4.9). На наше счастье, в Spring есть механизм выбора бина — он был рассмотрен в главе 3. Там вы узнали, что если в контексте Spring существует несколько бинов одного типа, то нужно сообщить Spring, какой из них следует внедрить. Для этого, как вы уже знаете, есть следующие способы:

- отметить аннотацией `@Primary` тот из бинов, который будет выбираться по умолчанию;
- присвоить бину имя с помощью аннотации `@Qualifier` и затем при DI обращаться к бину по этому имени.

Когда CommentService запрашивает зависимость типа CommentNotificationProxy, Spring должен выбрать, какую из нескольких существующих реализаций следует внедрить

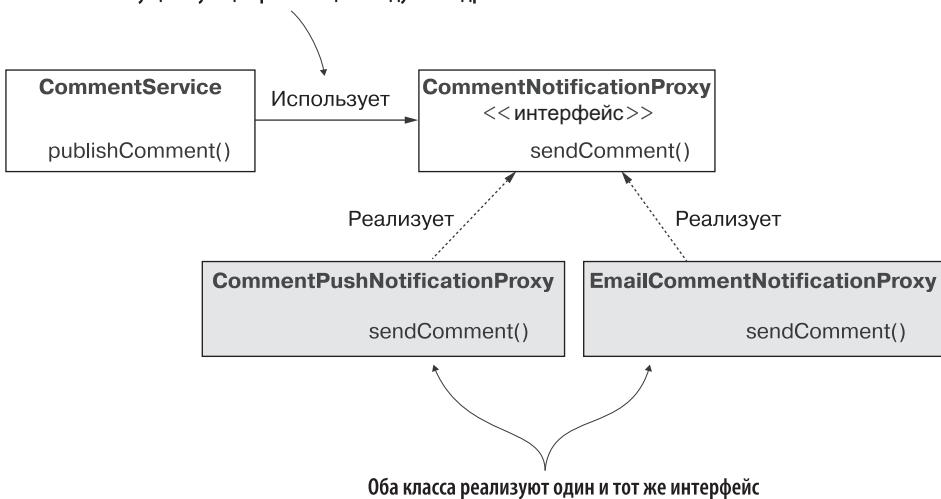


Рис. 4.9. Иногда в реальных приложениях встречается несколько реализаций одного интерфейса. При внедрении зависимостей через интерфейс необходимо сообщить Spring, какую именно из реализаций следует использовать

Теперь мы хотим убедиться, что эти два способа работают в том числе и с абстракциями. Добавим в наше приложение новый класс **CommentPushNotificationProxy** (который реализует интерфейс **CommentNotificationProxy**) и протестируем оба варианта по очереди, как показано в листинге 4.13. Чтобы все примеры хранились отдельно друг от друга, я создал новый проект **sq-ch4-ex5**, для начала взяв код из проекта **sq-ch4-ex2**.

Листинг 4.13. Новая реализация интерфейса CommentNotificationProxy

```

@Component
public class CommentPushNotificationProxy
    implements CommentNotificationProxy { ←
        Этот класс реализует интерфейс
        CommentNotificationProxy
    @Override
    public void sendComment(Comment comment) {
        System.out.println(
            "Sending push notification for comment: "
            + comment.getText());
    }
}
    
```

Если сейчас запустить приложение, то появится исключение, поскольку Spring не знает, какой из двух имеющихся в контексте бинов следует выбрать для внедрения. В следующем фрагменте кода показана наиболее интересная часть сообщения. В нем явно говорится о проблеме, с которой столкнулся Spring. Как

видим, исключение называется `NoUniqueBeanDefinitionException`, а уведомление гласит: `expected single matching but found 2` (ожидалось единственное соответствие, найдено 2). Так фреймворк сообщает, что ему нужно сказать более точно, какие бины из контекста требуется внедрить:

```
Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException:  
No qualifying bean of type 'proxies.CommentNotificationProxy' available:  
    expected single matching bean but found 2:  
  
commentPushNotificationProxy, emailCommentNotificationProxy
```

Использование аннотации `@Primary` для внедрения реализации по умолчанию

Самое простое решение задачи — использование аннотации `@Primary`. Нужно только поставить ее рядом с аннотацией `@Component`, чтобы реализация, описываемая этим классом, использовалась по умолчанию. Данный прием показан в листинге 4.14.

Листинг 4.14. Внедрение реализации, используемой по умолчанию, с помощью аннотации `@Primary`

```
@Component  
@Primary ←  
public class CommentPushNotificationProxy  
    implements CommentNotificationProxy {  
  
    @Override  
    public void sendComment(Comment comment) {  
        System.out.println(  
            "Sending push notification for comment: "  
            + comment.getText());  
    }  
}
```

Аннотация `@Primary` отмечает реализацию, которая используется по умолчанию при внедрении зависимостей

Благодаря этому небольшому изменению результат, выводимый приложением, выглядит гораздо понятнее. Обратите внимание: Spring действительно внедряет реализацию, описываемую новым классом:

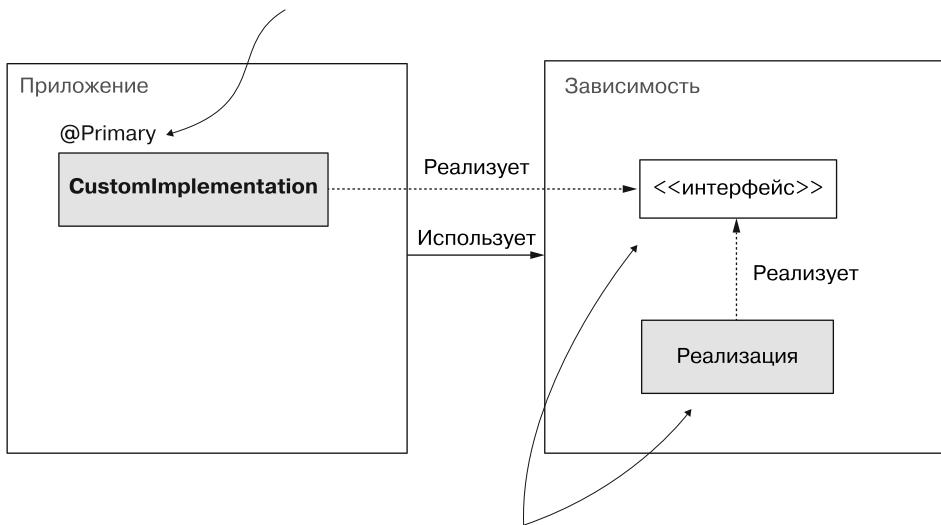
Storing comment: Demo comment	Spring внедряет новую реализацию, поскольку мы пометили ее аннотацией <code>@Primary</code>
Sending push notification for comment: Demo comment ←	

В этот момент мне часто задают вопрос: «Теперь у нас есть две реализации, но Spring всегда будет внедрять только одну из них? Зачем же тогда нужны два класса?»

Посмотрим, как вы можете попасть в подобную ситуацию при реальном сценарии. Как вы уже знаете, приложения бывают очень сложными, со множеством зависимостей. Возможно, вы будете использовать зависимость, которая предоставляет реализацию для определенного интерфейса (рис. 4.10), однако в какой-то момент

окажется, что эта реализация для вашего приложения не подходит. Тогда использование `@Primary` будет самым простым решением проблемы.

Вам нужно создать свою реализацию интерфейса, определяемого зависимостью. Но вам также нужно отметить ее как первичную, чтобы при использовании DI Spring внедрял именно вашу реализацию, а не ту, которая определяется зависимостью по умолчанию



В приложении есть зависимость. Она определяет интерфейс, а также его реализацию

Рис. 4.10. Иногда приходится использовать зависимости для интерфейсов, у которых уже есть реализации. И необходимо подключить аннотацию `@Primary`, чтобы ваша собственная реализация использовалась при DI по умолчанию. Тогда Spring будет знать, что нужно внедрять именно ее, а не ту, определенную зависимостью

Создание именованной реализации с помощью аннотации `@Qualifier` при внедрении зависимости

Иногда в приложениях необходимо определить несколько реализаций одного интерфейса, которые затем будут использоваться разными объектами. Предположим, что нам нужно создать две реализации для отправки сообщений о комментариях: по электронной почте и посредством push-уведомлений (рис. 4.11). Это реализации одного и того же интерфейса, однако они зависят от разных объектов приложения.

Чтобы протестировать данный подход, изменим код приложения. Этот вариант вы найдете в проекте `sq-ch4-ex6`. В следующем фрагменте кода показано, как создавать именованные реализации с помощью аннотации `@Qualifier`.

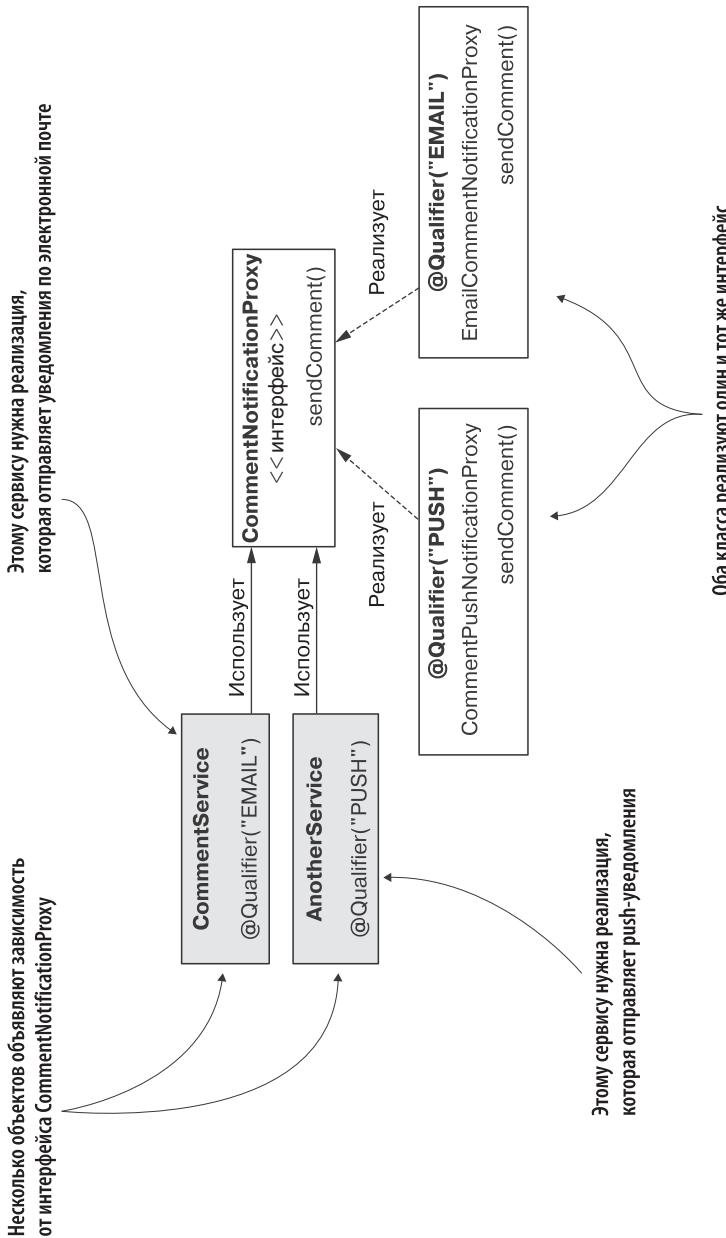


Рис. 4.11. Если объекты используют разные реализации одного контракта, то можно присвоить этим реализациям имена с помощью аннотации `@Qualifier`, и тогда Spring будет знать, когда и какую из них следует внедрить

Класс `CommentPushNotification` выглядит так:

```
@Component
@Qualifier("PUSH") ← С помощью аннотации @Qualifier присваиваем реализации имя PUSH
public class CommentPushNotificationProxy
    implements CommentNotificationProxy {
    // код класса
}
```

Класс `EmailCommentNotificationProxy` выглядит так:

```
@Component
@Qualifier("EMAIL") ← С помощью аннотации @Qualifier присваиваем реализации имя EMAIL
public class EmailCommentNotificationProxy
    implements CommentNotificationProxy {
    // код класса
}
```

Если нужно, чтобы Spring внедрил одну из реализаций, следует указать необходимое имя, снова использовав аннотацию `@Qualifier`. В листинге 4.15 показано, как внедрить выбранную реализацию в объект `CommentService` в качестве зависимости.

Листинг 4.15. Выбор внедряемой Spring реализации с помощью аннотации `@Qualifier`

```
@Component
public class CommentService {

    private final CommentRepository commentRepository;

    private final CommentNotificationProxy commentNotificationProxy;

    public CommentService(←
        CommentRepository commentRepository,
        @Qualifier("PUSH") CommentNotificationProxy
commentNotificationProxy) {
```

Каждый параметр, для которого нужно использовать специальную реализацию, сопровождается аннотацией @Qualifier

```
        this.commentRepository = commentRepository;
        this.commentNotificationProxy = commentNotificationProxy;
    }
    // остальной код класса
}
```

Зависимости, определенные аннотацией `@Qualifier`, будут внедрены Spring при запуске приложения. Посмотрим, что теперь появится в консоли:

```
Storing comment: Demo comment
Sending push notification for comment: Demo comment
```

Обратите внимание: Spring внедрил реализацию push-сообщений

4.3. ПОДРОБНЕЕ ОБ ОБЯЗАННОСТЯХ ОБЪЕКТОВ СО СТЕРЕОТИПНЫМИ АННОТАЦИЯМИ

До сих пор, рассматривая стереотипные аннотации, мы применяли в качестве примеров только `@Component`. Но вы обнаружите, что на практике для той же цели разработчики иногда используют и другие аннотации. Далее я покажу, как применяются еще две из них: `@Service` и `@Repository`.

В реальных проектах назначение компонента обычно указывают явно, используя стереотипную аннотацию. Как правило, это `@Component`. Но в таком случае вы не получаете никакой дополнительной информации об обязанностях реализуемого объекта, а для разработчиков они часто важны. Две такие обязанности мы рассмотрели в разделе 4.1 — это сервис и репозиторий.

Сервисы — объекты, задачей которых является реализация сценариев использования, а репозитории — объекты, отвечающие за сохранность данных. Поскольку эти обязанности являются типичными для большинства проектов и они очень важны при разработке классов, то разработчику, чтобы хорошо понимать структуру приложения, нужен способ явно их обозначать в проекте.

Чтобы отметить компонент, исполняющий обязанность сервиса, в Spring есть аннотация `@Service`. Для компонентов, реализующих задачи репозитория, используется аннотация `@Repository` (рис. 4.12). Все три аннотации, `@Component`, `@Service` и `@Repository`, являются стереотипными; встретив класс с подобной аннотацией, Spring создает экземпляр такого класса и добавляет этот экземпляр в контекст.

В примерах, приведенных в этой главе, вместо аннотации `@Component` для класса `CommentService` можно было бы использовать `@Service`. Таким образом мы бы явно обозначили обязанность объекта и сделали бы ее более заметной для разработчиков, которые впоследствии будут читать код класса. В следующем примере показан класс `CommentService` со стереотипной аннотацией `@Service`:

```
@Service
public class CommentService {
    // код класса
}
```

С помощью аннотации `@Service` мы показываем, что этот объект является компонентом, исполняющим обязанность сервиса

Аналогичным образом можно обозначить обязанность класса репозитория с помощью аннотации `@Repository`:

```
@Repository
public class DBCCommentRepository implements CommentRepository {
    // код класса
}
```

С помощью аннотации `@Repository` мы показываем, что этот объект является компонентом, исполняющим обязанность репозитория

Данный пример (`sq-ch4-ex7`) вы найдете среди проектов, прилагаемых к книге.

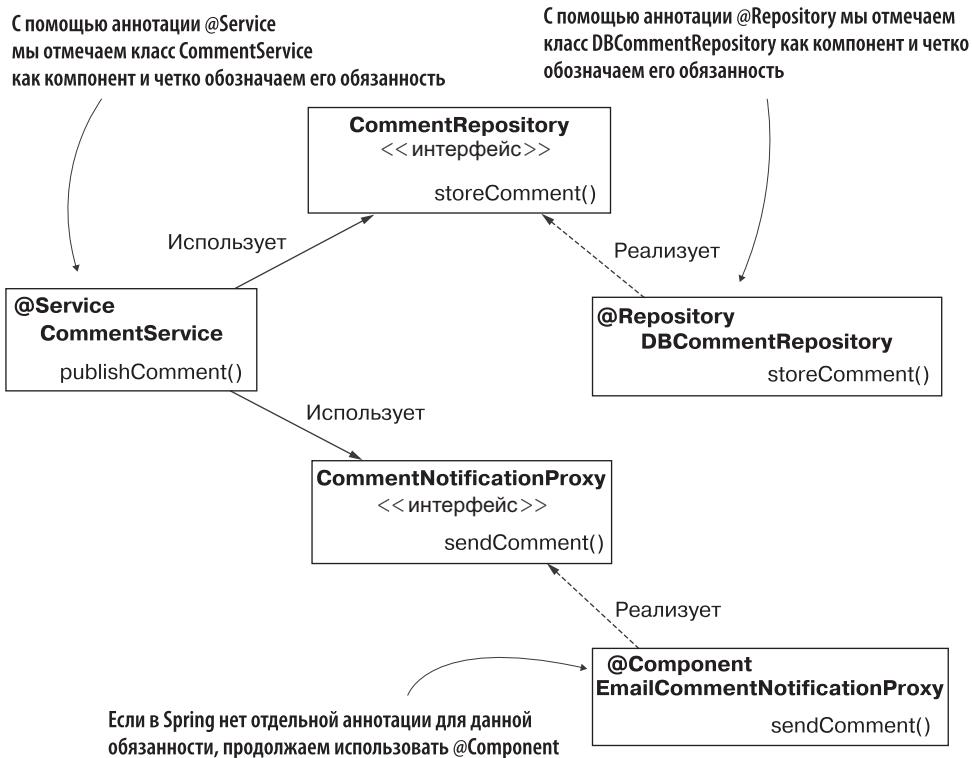


Рис. 4.12. Аннотации `@Service` и `@Repository` позволяют явно указать обязанности компонентов в структуре классов. Если в Spring нет специальной аннотации для данной обязанности, продолжаем использовать `@Component`

РЕЗЮМЕ

- Разделение реализаций посредством абстракций — хороший тон проектирования классов. Разделение объектов позволяет легко изменять реализации, не затрагивая при этом слишком много частей приложения, благодаря чему приложение проще расширять и поддерживать.
- В Java для разделения реализаций применяются интерфейсы. Принято также говорить, что посредством интерфейсов создаются контракты между реализациями.
- При использовании абстракции с внедрением зависимостей Spring знает, где искать бин, который ее реализует.
- Если нужно создать экземпляр класса и поместить его в контекст Spring, то такой класс сопровождается стереотипной аннотацией. Стереотипные аннотации никогда не применяются к интерфейсам.

- Если в контексте Spring есть несколько бинов, представляющих собой разные реализации одной и той же абстракции, существует несколько способов обозначить для Spring, какой из этих бинов следует внедрить:
 - отметить один из бинов как используемый по умолчанию с помощью аннотации `@Primary`;
 - создать именованный бин с помощью аннотации `@Qualifier` и затем дать Spring инструкцию внедрить именно этот бин, указав его имя.
- Для компонентов с обязанностями сервиса вместо стереотипной аннотации `@Component` можно применить аннотацию `@Service`. Аналогичным образом для компонентов с обязанностью репозитория вместо `@Component` можно использовать аннотацию `@Repository`. Таким образом мы явно обозначаем обязанность компонента, благодаря чему структура классов становится более понятной и удобной для чтения.



Контекст Spring: области видимости и жизненный цикл бинов

В этой главе

- ✓ Бины с одиночной областью видимости.
- ✓ Немедленное и «ленивое» создание одиночных бинов.
- ✓ Бины с прототипной областью видимости.

Ранее мы уже рассмотрели несколько основных моментов, касающихся бинов (экземпляров объектов, находящихся под управлением Spring). Мы изучили важные особенности синтаксиса, которые необходимо знать, чтобы создавать бины, а также узнали, как создаются связи между бинами (в том числе поняли, зачем нужны абстракции). Но мы пока не затрагивали вопрос, когда и как Spring создает бины. До сих пор мы просто позволяли фреймворку действовать по умолчанию.

Я решил не поднимать эту тему в самом начале книги, потому что хотел обратить ваше внимание на синтаксис, который вам сразу понадобится в проектах. Однако сценарии работы реальных приложений очень сложны, и иногда того, что делает фреймворк по умолчанию, бывает недостаточно. Поэтому в данной главе мы глубже рассмотрим, каким образом Spring управляет объектами в контексте.

В Spring предусмотрено несколько вариантов создания бинов и управления их жизненным циклом — в мире фреймворка эти способы называются *областями видимости* (scopes). Далее будут рассмотрены две области видимости, которые вам будут часто встречаться в приложениях Spring: *одиночка* и *прототип*.

ПРИМЕЧАНИЕ

Позже, в главе 9, вы познакомитесь еще с несколькими областями видимости бинов, которые встречаются в веб-приложениях: это запрос, сессия и приложение.

Одиночная область видимости — это область видимости, которую бин Spring получает по умолчанию. Именно ее мы использовали до сих пор. В разделе 5.1 мы начнем именно с нее. Прежде всего мы рассмотрим, как именно Spring управляет одиночными бинами, а затем обсудим несколько важных моментов, которые необходимо знать об использовании одиночных областей видимости в реальных приложениях.

В разделе 5.2 мы перейдем к прототипным бинам. Мы обратим внимание на то, чем прототипная область видимости отличается от одиночной и в каких ситуациях следует выбирать ту или иную из них.

5.1. ИСПОЛЬЗОВАНИЕ ОДИНОЧНОЙ ОБЛАСТИ ВИДИМОСТИ

Одиночная область видимости бина — вариант, выбираемый Spring по умолчанию для управления бинами в контексте. Именно она будет встречаться вам чаще всего в реальных приложениях. В подразделе 5.1.1 вы узнаете, как Spring создает одиночные бины и контролирует их — это исключительно важно для понимания, в каких случаях их следует использовать. Мы рассмотрим два примера с разными способами определения бинов (о которых вы узнали в главе 2) и проанализируем поведение Spring в этих случаях. Затем (в подразделе 5.1.2) обсудим важнейшие аспекты использования одиночных бинов в реальных приложениях. В завершение раздела вы познакомитесь с двумя способами создания одиночных бинов (немедленным и «ленивым»), мы обсудим, когда их следует применять.

5.1.1. Что такое одиночный бин

Вначале рассмотрим поведение Spring в случае бинов с одиночной областью видимости. Вам следует понять, чего можно ожидать при ее использовании, особенно учитывая, что одиночная область видимости бина применяется в Spring по умолчанию (и встречается чаще всего). В этом разделе я покажу, как код, который вы пишете, влияет на контекст Spring, чтобы вам было легче понимать поведение фреймворка. Затем мы проверим это поведение на нескольких примерах.

Spring создает одиночные бины при загрузке контекста и присваивает им имена (иначе, ID бина). Эта область видимости называется одиночной, поскольку при обращении к конкретному бину мы всегда получаем один и тот же экземпляр.

Но будьте внимательны! В контексте Spring может существовать несколько экземпляров одного типа, но с разными именами. Я особенно акцентирую этот момент. Вероятно, для обозначения шаблонов проектирования вам уже встречалось слово «одиночка» (синглтон). Если нет, то вы не спутаете его с одиночной областью видимости бинов — можете пропустить следующий абзац.

Однако если синглтон как шаблон проектирования вам все же знаком, одиночная область видимости бинов в Spring может показаться вам странной, ведь одиночный шаблон предполагает наличие единственного экземпляра данного типа в приложении. В Spring таких экземпляров может быть несколько, а само понятие «одиночный» означает уникальность имени, но не уникальность наличия в пределах приложения (рис. 5.1).

Объявление бинов с одиночной областью видимости с помощью аннотации @Bean

Рассмотрим поведение бина с одиночной областью видимости на примере. Мы добавим экземпляр в контекст Spring с помощью аннотации `@Bean`, а затем просто обратимся к нему несколько раз из класса `Main`. Таким образом мы докажем, что при каждой отсылке к бину мы всегда получаем один и тот же экземпляр.

На рис. 5.2 визуально представлена связь между контекстом и кодом, определяющим конфигурацию этого контекста. Кофейное зерно на рисунке олицетворяет экземпляр, который Spring добавляет в контекст. Обратите внимание, что в контексте есть только один экземпляр (одно кофейное зерно) с присвоенным именем. Как уже говорилось в главе 2, при добавлении бина в контекст с помощью аннотации `@Bean` именем бина становится имя метода, сопровождаемого данной аннотацией.

В настоящем примере я добавил бин в контекст с помощью аннотации `@Bean`. Но мне бы не хотелось, чтобы у вас сложилось мнение, будто одиночные бины создаются только так. Если бы для добавления бина в контекст я воспользовался стереотипной аннотацией (такой как `@Component`), результат был бы таким же. Далее мы в этом убедимся.

Также обратите внимание, что при получении бина из контекста Spring я явно использовал имя бина. Как вы уже знаете из главы 2, если в контексте есть только один бин данного типа, обращаться к нему по имени не нужно. Можно это сделать и по его типу. Я использовал имя, только чтобы подчеркнуть, что мы имеем в виду один и тот же бин. Я мог бы вызвать бин и по его типу — в обоих случаях мы бы получили ссылку на один и тот же (единственный) имеющийся в контексте экземпляр `CommentService`.

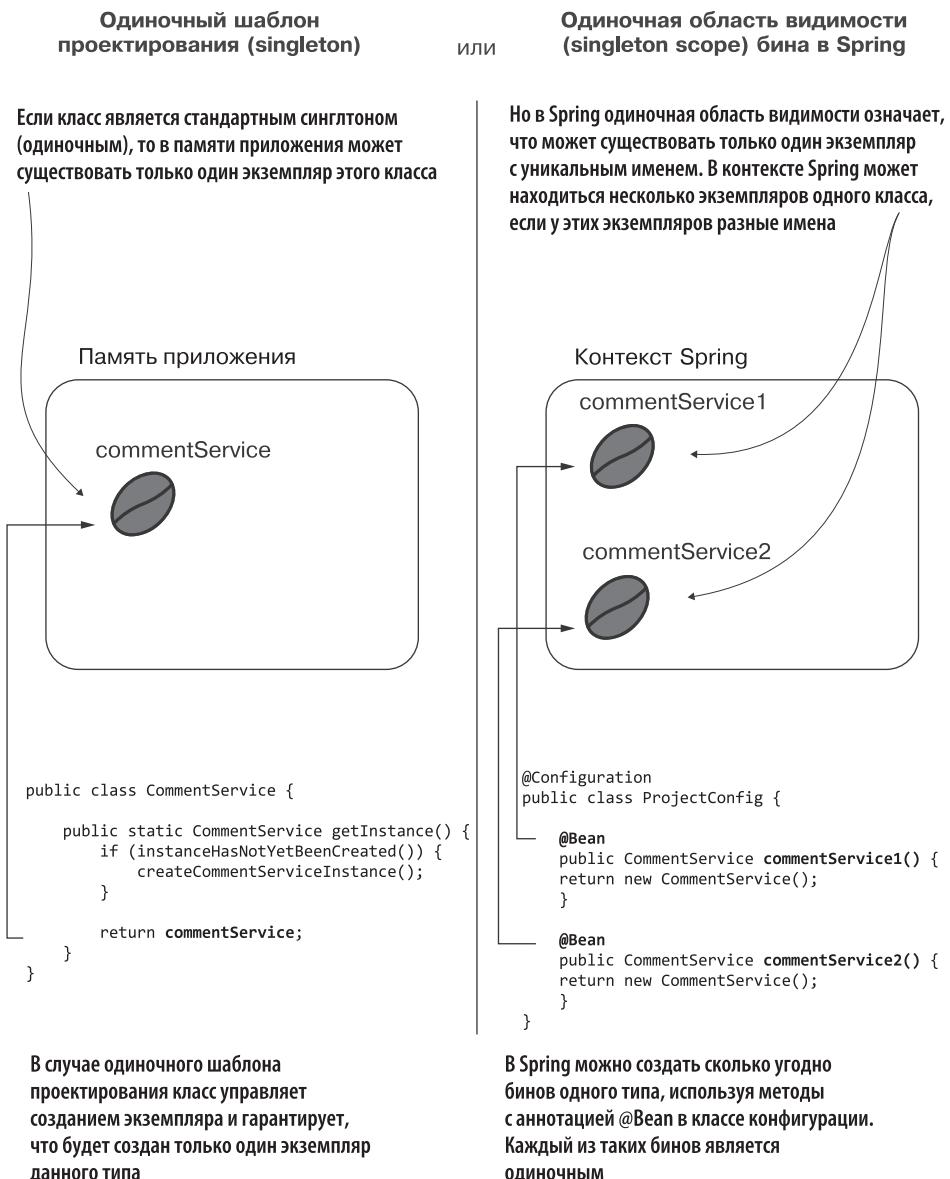


Рис. 5.1. Говоря об одиночном классе в приложении, мы имеем в виду, что есть только один экземпляр в классе и класс управляет созданием этого экземпляра. Но в Spring одиночная область видимости не означает, что в контексте есть только один экземпляр данного типа. Это значит лишь, что конкретному экземпляру присвоено имя и при обращении по этому имени всегда будет возвращаться один и тот же экземпляр

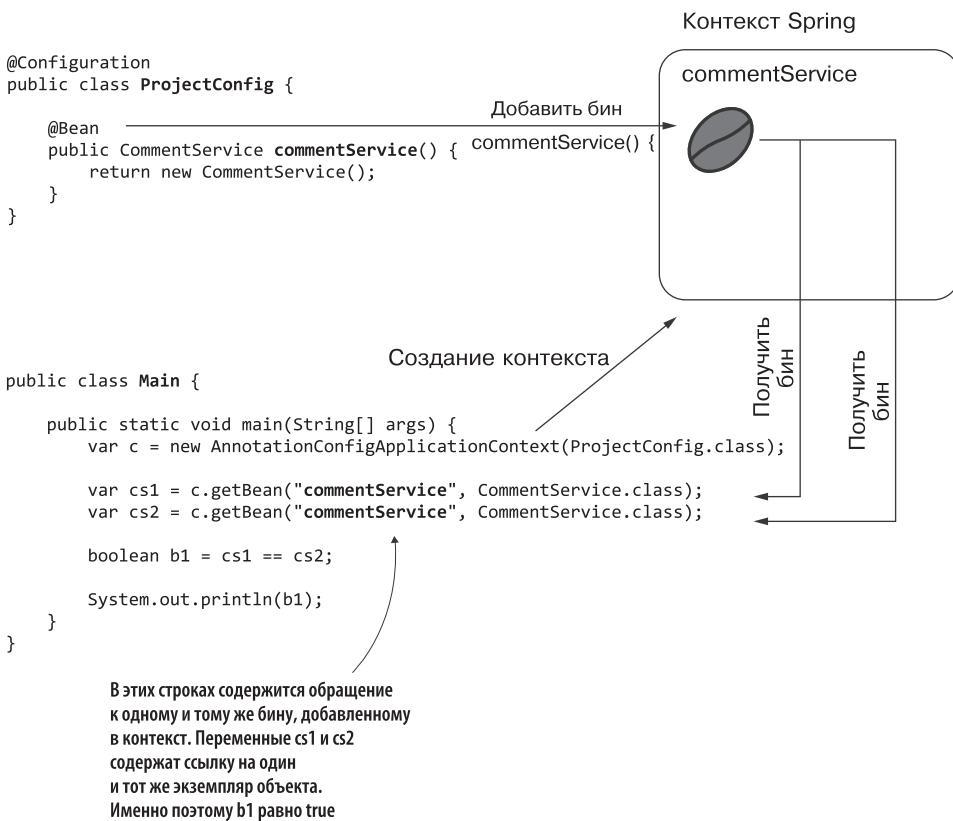


Рис. 5.2. Одиночный бин. При запуске приложение инициализирует контекст и добавляет туда бин. В данном случае мы объявили бин с помощью аннотации `@Bean`, так что его идентификатором стало имя метода. Каждый раз, используя это название, мы получим ссылку на один и тот же экземпляр

Для завершения напишем код и выполним его. Этот пример вы найдете в проекте `sq-ch5-ex1`. Нам нужно определить пустой класс `CommentService`, как показано в следующем фрагменте кода, а затем написать класс конфигурации и класс `Main`, как на рис. 5.2:

```
public class CommentService { }
```

В листинге 5.1 представлено определение класса конфигурации, в котором для добавления экземпляра типа `CommentService` в контекст Spring использован метод с аннотацией `@Bean`.

Листинг 5.1. Добавление бина в контекст Spring

```
@Configuration
public class ProjectConfig {

    @Bean ← Добавление бина CommentService в контекст Spring
    public CommentService commentService() {
        return new CommentService();
    }
}
```

В листинге 5.2 представлен класс `Main`, с помощью которого мы тестируем поведение Spring в случае одиночного бина. Мы дважды получаем ссылку на бин `CommentService` и рассчитываем, что каждый раз ссылка будет одна и та же.

Листинг 5.2. Класс `Main`, используемый для тестирования поведения Spring при внедрении одиночного бина

```
public class Main {

    public static void main(String[] args) {
        var c = new AnnotationConfigApplicationContext(ProjectConfig.class);

        var cs1 = c.getBean("commentService", CommentService.class);
        var cs2 = c.getBean("commentService", CommentService.class);

        boolean b1 = cs1 == cs2; ←
        System.out.println(b1);   | Поскольку обе переменные содержат
                                | одну и ту же ссылку, результатом
                                | операции является true
    }
}
```

При запуске приложения в консоль будет выведено `true`, поскольку Spring, имея дело с одиночным бином, всегда возвращает одну и ту же ссылку.

Создание одиночных бинов с помощью стереотипных аннотаций

Как уже говорилось ранее, поведение Spring по отношению к одиночным бинам не зависит от того, были ли они созданы с помощью аннотации `@Bean` или какой-либо из стереотипных аннотаций. Но я хочу еще ярче продемонстрировать это на следующем примере. Рассмотрим структуру классов, в которой два класса сервисов зависят от одного и того же репозитория. Предположим, что классы `CommentService` и `UserService` зависят от репозитория `CommentRepository`, как на рис. 5.3.

В обычной структуре классов сценарии
использования реализуются классами сервисов Класс репозитория отвечает за сохранение данных

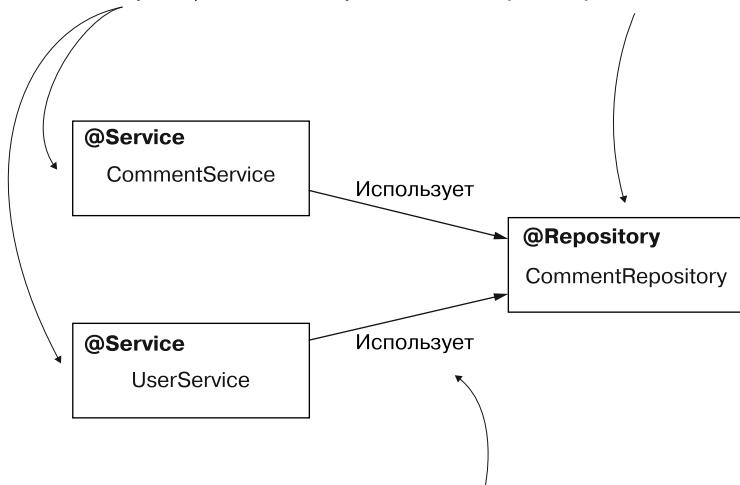


Рис. 5.3. Структура классов для выбранного сценария. Два класса сервисов нуждаются в репозитории для выполнения своих сценариев использования. Если реализовать эти сервисы как одиночные бины, то в контексте Spring будет создано по одному экземпляру для каждого из этих классов

Не имеет значения, почему эти классы зависят друг от друга. Наши сервисы ничего не делают (это просто сценарий). Предположим, что данная структура является частью более сложного приложения, и обратим внимание на связи между бинами и на то, каким образом Spring устанавливает связи в контексте. На рис. 5.4 представлена визуализация контекста параллельно с кодом, в котором содержится его конфигурация.

Убедимся, что поведение Spring именно таково, каким мы его ожидаем увидеть. Для этого создадим три класса, сравним ссылки, которые Spring будет внедрять в бины сервисов, — и увидим, что ссылка выдается одна и та же. В следующем фрагменте кода представлено определение класса `CommentRepository` (проект `sq-ch5-ex2`):

```
@Repository
public class CommentRepository { }
```

В следующем фрагменте кода представлено определение класса `CommentService`. Обратите внимание, что для внедрения экземпляра типа `CommentRepository` в атрибут, объявленный внутри класса, я использовал аннотацию `@Autowired`.

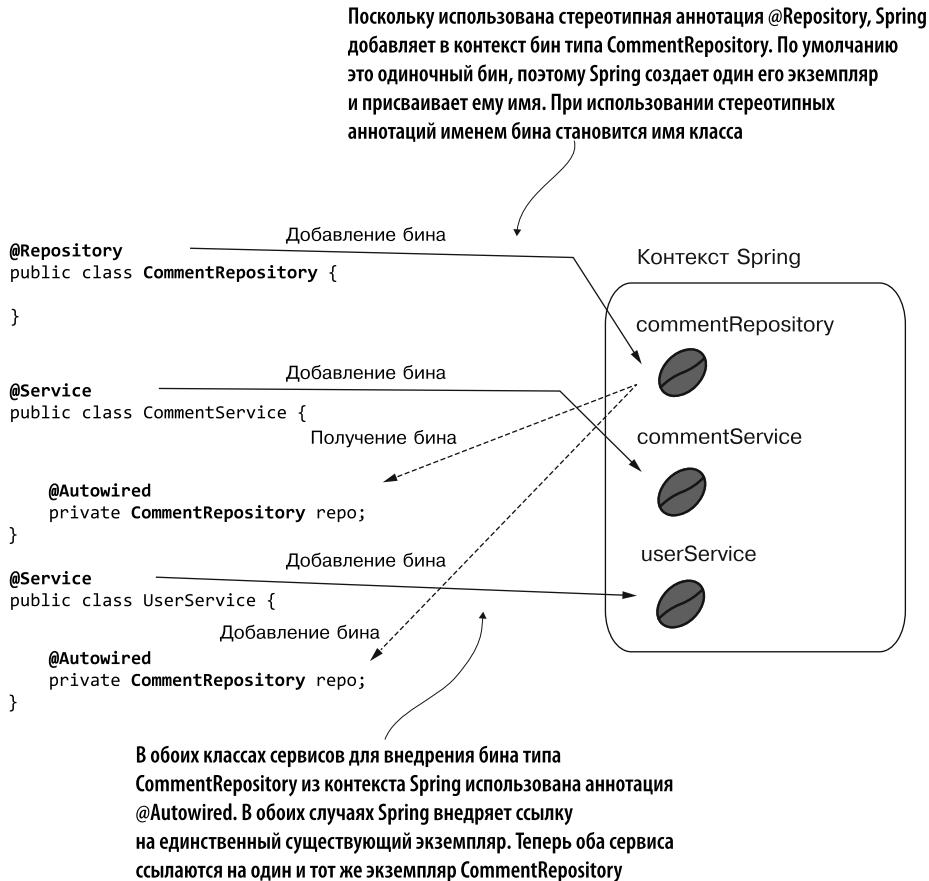


Рис. 5.4. Бины, созданные с использованием стереотипных аннотаций, также являются одиночными. Если ссылка на бин внедряется с помощью аннотации `@Autowired`, Spring внедряет ссылку на одиничный бин везде, где это требуется

Я определил также геттер, который собираюсь использовать позднее, чтобы показать, что Spring внедряет в оба бина сервиса ссылку на один и тот же объект:

```

@Service
public class CommentService {

    @Autowired
    private CommentRepository commentRepository;

    public CommentRepository getCommentRepository() {
        return commentRepository;
    }
}

```

Следуя той же логике, что и в случае с `CommentService`, определим класс `UserService`:

```
@Service
public class UserService {

    @Autowired
    private CommentRepository commentRepository;

    public CommentRepository getCommentRepository() {
        return commentRepository;
    }
}
```

В отличие от первого примера, в этом проекте класс конфигурации пуст. Нам нужно только сообщить Spring, где находятся классы со стереотипными аннотациями. Как уже говорилось в главе 2, для этого используется аннотация `@ComponentScan`. Определение класса конфигурации представлено в следующем фрагменте кода:

```
@Configuration
@ComponentScan(basePackages = {"services", "repositories"})
public class ProjectConfig {

}
```

В классе `Main` мы получаем ссылки на два сервиса и сравниваем их зависимости, чтобы убедиться, что Spring внедрил в оба сервиса один и тот же экземпляр. Класс `Main` представлен в листинге 5.3.

Листинг 5.3. Класс Main: проверка поведения Spring при внедрении одиночного бина

```
public class Main {
    public static void main(String[] args) {
        var c = new AnnotationConfigApplicationContext(
            ProjectConfig.class);

        var s1 = c.getBean(CommentService.class);
        var s2 = c.getBean(UserService.class);

        boolean b = ←
            s1.getCommentRepository() == s2.getCommentRepository();

        System.out.println(b); ←
    }
}
```

The code is annotated with several callouts explaining its behavior:

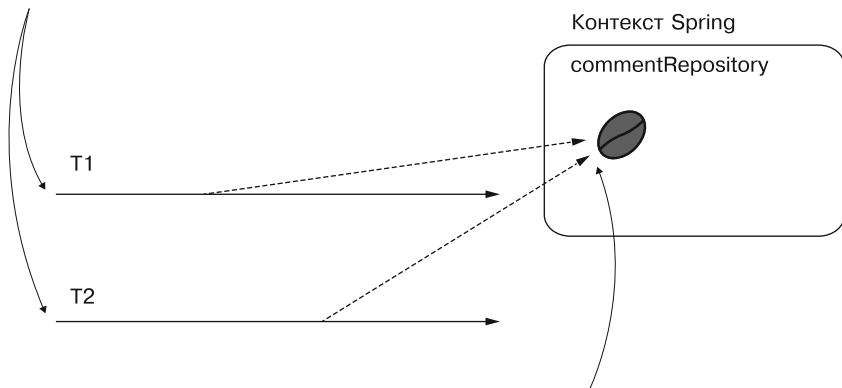
- A callout points to the creation of the Spring context: "Создание контекста Spring на основании класса конфигурации".
- A callout points to the retrieval of beans from the context: "Получение ссылок на два бина сервисов из контекста Spring".
- A callout points to the comparison of repository references: "Сравнение внедренных Spring ссылок на зависимость репозитория".
- A callout points to the final print statement: "Поскольку зависимость (CommentRepository) является одиночным бином, оба сервиса содержат одну и ту же ссылку, так что в строке всегда будет выводиться true".

5.1.2. Одиночные бины в реальных приложениях

До сих пор мы рассматривали, как Spring управляет бинами с одиночной областью видимости. Пора обратить внимание на важные аспекты работы с одиночными бинами. Для начала разберем несколько сценариев, в которых следует или, наоборот, не стоит использовать одиночные бины.

Поскольку область видимости одиночного бина предполагает, что доступ к экземпляру объекта имеют несколько компонентов приложения, главное, что следует учитывать, — эти бины должны быть неизменяемыми. Как правило, в реальных приложениях (таких как веб-приложения) операции выполняются в нескольких потоках. Следовательно, разные потоки могут иметь доступ к одному экземпляру объекта. Если эти потоки изменят экземпляр, то попадут в *состояние гонки* (рис. 5.5).

Эти стрелки обозначают последовательность выполнения двух потоков — T1 и T2



Если два потока попытаются одновременно получить доступ к одному экземпляру и изменить его, то возникнет состояние гонки

Рис. 5.5. Когда несколько потоков пытаются получить доступ к одиночному бину, они обращаются к одному и тому же экземпляру. Если эти потоки одновременно попытаются изменить его, возникнет состояние гонки. Если бин не создан с учетом конкурентности, то состояние гонки приведет к неожиданным результатам или исключениям при выполнении приложения

Состояние гонки — это ситуация, которая возникает в многопоточных архитектурах, когда несколько потоков пытаются изменить общедоступный ресурс. При появлении данной проблемы разработчику следует тщательно синхронизировать потоки во избежание ошибок или неожиданных результатов выполнения приложения.

Если вам нужны изменяемые одиночные бины (атрибуты которых можно модифицировать), необходимо сделать их согласованными (для этого обычно используют синхронизацию потоков). Но одиночные бины не рассчитаны на то, чтобы быть конкурентными. Они, как правило, используются для определения базовой структуры классов и делегирования обязанностей между собой. Технически синхронизация одиночных бинов возможна, но это не является хорошим тоном в программировании. Синхронизация потоков для конкурентности экземпляра резко снижает производительность приложения. К счастью, есть другие способы решить эту проблему.

Помните, в главе 3 говорилось, что хорошим тоном в программировании является скорее использование конструктора DI, нежели внедрение в поле? Одним из преимуществ внедрения в конструктор является возможность сделать экземпляры неизменяемыми (пометив поля бина как `final`). В предыдущем примере можно улучшить определение класса `CommentService`, заменив внедрение в поле внедрением в конструктор следующим образом:

```
@Service
public class CommentService {
    private final CommentRepository commentRepository; ←

    public CommentService(CommentRepository commentRepository) {
        this.commentRepository = commentRepository;
    }

    public CommentRepository getCommentRepository() {
        return commentRepository;
    }
}
```

Если отметить поле как `final`, то всем будет ясно, что это поле не должно изменяться

ТРИ ГЛАВНЫХ ПРАВИЛА ИСПОЛЬЗОВАНИЯ БИНОВ

- Размещайте объект в контексте Spring в виде бина только в том случае, если необходимо, чтобы Spring управлял этим объектом и мог расширить его возможности за счет своего функционала. Если объект не нуждается в возможностях, предоставляемых фреймворком, то не следует делать его бином.
- Если нужно сделать объект бином и разместить его в контексте Spring, то бин должен быть одиночным только в случае, если он неизменяемый. Постарайтесь не создавать изменяемые одиночные бины.
- Если бин все же должен быть изменяемым, лучше использовать прототипную область видимости, которая будет описана в разделе 5.2.

5.1.3. Немедленное и «ленивое» создание экземпляров

Как правило, Spring создает одиночные бины при инициализации контекста — таково поведение фреймворка по умолчанию (его также называют *немедленным* созданием экземпляров). Данный способ мы использовали до сих пор. Далее мы рассмотрим другой предусмотренный фреймворком вариант, «*ленивое*» создание экземпляров, и сравним его с первой возможностью. В «ленивом» случае Spring не создает одиночных бинов при запуске контекста. Вместо этого каждый экземпляр создается в момент, когда происходит первое обращение к данному бину. Чтобы проследить разницу между этими двумя способами, рассмотрим их на примере, а затем обсудим преимущества и недостатки их применения в реальных приложениях.

Чтобы протестировать вариант по умолчанию (немедленный), в исходном сценарии нам нужен только один бин. Данный сценарий реализован в проекте sq-ch5-ex3. Я оставил все существующие имена без изменений, так что класс будет по-прежнему называться `CommentService`. На его основе создайте бин, используя либо аннотацию `@Bean` (как показано в следующем фрагменте кода), либо стереотипную аннотацию. Но в любом случае обязательно добавьте в конструкторе класса вывод в консоль. Таким образом, если фреймворк вызовет конструктор класса, мы сразу же это увидим:

```
@Service  
public class CommentService {  
  
    public CommentService() {  
        System.out.println("CommentService instance created!");  
    }  
}
```

Если вы использовали стереотипную аннотацию, не забудьте добавить в класс конфигурации аннотацию `@ComponentScan`. Мой класс конфигурации выглядит так:

```
@Configuration  
@ComponentScan(basePackages = {"services"})  
public class ProjectConfig {  
  
}
```

В классе `Main` мы только создаем экземпляр контекста Spring. Обратите внимание на то, что никакой объект не использует бин `CommentService`, однако Spring все равно создаст данный экземпляр и сохранит его в контексте. Мы узнаем об этом при запуске приложения, поскольку увидим в консоли вывод, генерируемый конструктором `CommentService`. Класс `Main` представлен в следующем фрагменте кода:

```
public class Main {
    public static void main(String[] args) { ←
        var c = new AnnotationConfigApplicationContext(ProjectConfig.class);
    }
}
```

В этом приложении создается контекст Spring,
но бин CommentService нигде не используется

Несмотря на то что бин нигде не используется, при запуске приложения вы увидите в консоли следующий текст:

```
CommentService instance created!
```

Теперь внесем изменения в наш пример (проект sq-ch5-ex4), добавив аннотацию `@Lazy` перед классом (в случае стереотипных аннотаций) или перед методом с `@Bean` (в случае использования метода с аннотацией `@Bean`). Запустив приложение, вы увидите, что сообщение о создании бина в консоли больше не появляется. При заданной конфигурации Spring будет создавать бин, только если он будет использоваться, а в нашем примере бин `CommentService` не используется.

```
@Service
@Lazy ←
public class CommentService {
    public CommentService() {
        System.out.println("CommentService instance created!");
    }
}
```

Аннотация `@Lazy` сообщает Spring, что бин нужно создавать
только тогда, когда он в первый раз кому-нибудь понадобится

Теперь изменим класс `Main` — добавим туда ссылку на бин `CommentService`:

```
public class Main {

    public static void main(String[] args) {
        var c = new AnnotationConfigApplicationContext(ProjectConfig.class);

        System.out.println("Before retrieving the CommentService");
        var service = c.getBean(CommentService.class); ←
        System.out.println("After retrieving the CommentService");
    }
}
```

В этой строке Spring должен предоставить ссылку на бин
CommentService, и поэтому Spring здесь его создает

Теперь, запустив приложение, мы снова увидим текст в консоли. Фреймворк создает бин только тогда, когда он нужен:

```
Before retrieving the CommentService
CommentService instance created!
After retrieving the CommentService
```

Когда предпочтительнее использовать немедленный, а когда — «ленивый» вариант работы? Как правило, удобнее всего создать все экземпляры в самом начале, одновременно с контекстом (немедленно); в этом случае, если один экземпляр

делегирует функции другому, второй бин гарантированно уже будет существовать. При «ленивом» же создании бина фреймворк должен сначала проверить, существует ли данный экземпляр, и, если нет, создать его.

Таким образом, с точки зрения производительности лучше заранее создать все экземпляры, которые должны находиться в контексте (немедленно), иначе при делегировании функций от одного бина другому фреймворк должен будет сделать еще несколько дополнительных проверок. К тому же, если что-либо пойдет не так и фреймворк не сможет создать бин, вы сразу это заметите при запуске приложения. При «ленивом» варианте проблема может быть обнаружена только в уже работающем приложении, когда возникнет необходимость в создании конкретного бина.

Однако «ленивое» создание экземпляров не всегда зло. Некоторое время назад я работал над большим монолитным приложением. Оно устанавливалось на разных устройствах, где клиенты использовали его для разных целей. Как правило, каждому пользователю нужна была лишь малая часть функционала приложения, так что создание сразу всех бинов одновременно с контекстом Spring требовало неоправданно большого объема памяти. Поэтому большинство бинов приложения разработчики сделали «ленивыми», чтобы каждый раз создавались только необходимые экземпляры.

И все же мой совет — выбирать вариант по умолчанию, то есть немедленное создание экземпляров. Этот способ, как правило, имеет больше преимуществ. А если вдруг вы окажетесь в ситуации, подобной описанному выше случаю, сначала проверьте, нельзя ли как-либо изменить структуру приложения. Например, в моей истории было бы лучше, если бы приложение имело модульную структуру или было представлено в виде микросервисов. Такая архитектура позволила бы разработчикам развертывать только ту часть приложения, которая нужна конкретным клиентам, и необходимость в «ленивом» создании бинов отпала бы. Но на практике реализовать такое бывает сложно из-за ограничений по времени или стоимости. Тем не менее, если невозможно устраниТЬ настоящую причину болезни, иногда можно хотя бы избавиться от некоторых симптомов.

5.2. ПРОТОТИПНАЯ ОБЛАСТЬ ВИДИМОСТИ БИНОВ

Рассмотрим вторую область видимости бинов в Spring — прототипную. Как вы узнаете далее, иногда вместо одиночных бинов лучше использовать бины с прототипной областью видимости. В подразделе 5.2.1 мы рассмотрим поведение фреймворка по отношению к таким бинам. Затем на нескольких примерах вы научитесь изменять область видимости бина на прототипную. Наконец, в подразделе 5.2.2 вы узнаете, в каких ситуациях следует использовать прототипную область видимости.

5.2.1. Как работают прототипные бины

Прежде чем обсуждать, в каких практических ситуациях следует использовать прототипные бины, разберемся, что делает Spring, чтобы ими управлять. Как вы скоро убедитесь, идея здесь очень проста. Каждый раз, когда запрашивается ссылка на прототипный бин, Spring создает новый экземпляр объекта. В случае прототипных бинов Spring не создает сам объект и не управляет им. Фреймворк управляет только типом объекта и создает новый экземпляр всякий раз, когда какой-либо объект запрашивает ссылку на этот бин. На рис. 5.6 я изобразил бин в виде кофейного дерева (чтобы получить свежие зерна, их нужно снять с дерева — подобно этому, всякий раз, когда нужен бин, требуется создать новый экземпляр). Мы будем по-прежнему употреблять термин «бин», но данная растительная аналогия поможет вам быстро понять и запомнить поведение Spring относительно прототипных бинов.

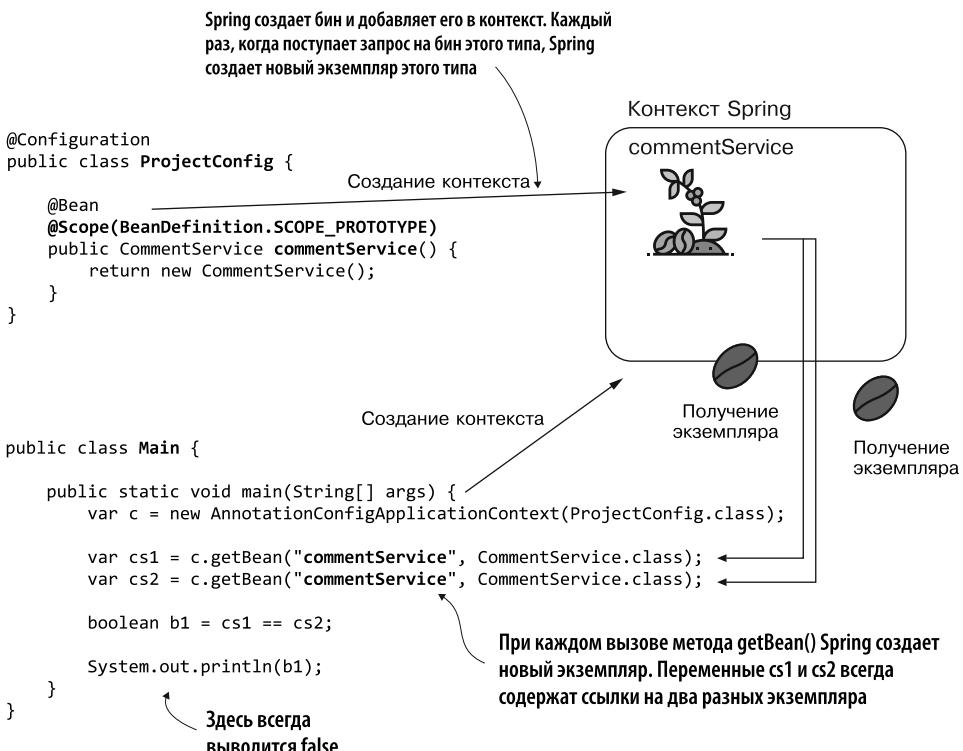


Рис. 5.6. С помощью аннотации `@Scope` мы изменили область видимости бина на прототипную. Здесь он представлен в виде кофейного дерева, ведь всякий раз, когда нужно получить бин, создается новый экземпляр объекта. Поэтому переменные `cs1` и `cs2` всегда содержат разные ссылки, так что в консоль всегда выводится `false`

Как видно на рис. 5.6, для изменения области видимости бина нам понадобится новая аннотация — `@Scope`. Она ставится вместе с `@Bean` перед методом, объявляющим бин. При объявлении бина с помощью стереотипных аннотаций `@Scope` вместе со стереотипными аннотациями ставится перед классом, в котором объявляется бин.

Применение прототипных бинов решает проблемы конкурентности, поскольку каждый поток получает свой экземпляр, так что определение нескольких прототипных бинов не создает проблем (рис. 5.7).

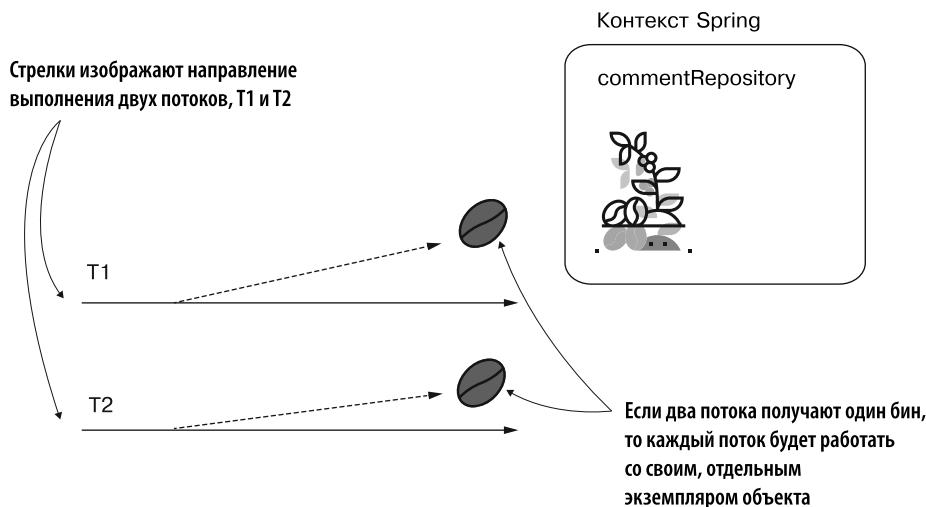


Рис. 5.7. Если несколько потоков запрашивают один прототипный бин, то каждый поток получает свой экземпляр. Благодаря этому потоки не попадают в состояние гонки

Объявление бинов с прототипной областью видимости с помощью аннотации `@Bean`

Для наглядности мы напишем проект `sq-ch5-ex5` и посмотрим, как Spring управляет прототипными бинами. Мы создадим бин `CommentService` и объявим его прототипным, чтобы при каждом его запросе получать новый экземпляр. Класс `CommentService` выглядит так:

```
public class CommentService {  
}
```

В классе конфигурации мы определим бин класса `CommentService`, как показано в листинге 5.4.

Листинг 5.4. Объявление прототипного бина в классе конфигурации

```
@Configuration
public class ProjectConfig {

    @Bean
    @Scope(BeanDefinition.SCOPE_PROTOTYPE) ←———— Объявляем этот бин прототипным
    public CommentService commentService() {
        return new CommentService();
    }
}
```

Чтобы убедиться, что при каждом запросе мы получаем новый экземпляр, создадим класс `Main` и дважды запросим этот бин из контекста. Мы увидим, что ссылки будут разными. Определение класса `Main` представлено в листинге 5.5.

Листинг 5.5. Класс Main с проверкой поведения Spring по отношению к прототипным бинам

```
public class Main {

    public static void main(String[] args) {
        var c = new AnnotationConfigApplicationContext(ProjectConfig.class);

        var cs1 = c.getBean("commentService", CommentService.class);
        var cs2 = c.getBean("commentService", CommentService.class);

        boolean b1 = cs1 == cs2; ←———— Переменные cs1 и cs2 содержат
                               ссылки на разные экземпляры
        System.out.println(b1); ←———— В консоль всегда
                               выводится false
    }
}
```

Запустив приложение, вы увидите, что в консоли всегда выводится `false`. Этот результат подтверждает, что при вызове метода `getBean()` возвращаются два разных экземпляра.

Объявление бинов с прототипной областью видимости посредством стереотипных аннотаций

Создадим также проект `sq-ch5-ex6`, чтобы проследить поведение прототипных бинов, полученных посредством автомонтажа. Мы определим прототипный бин `CommentRepository` и внедрим его в два других, сервисных бина посредством аннотации `@Autowired`. Вы увидите, что каждый бин сервиса получает ссылку на свой экземпляр `CommentRepository`. Такая схема поведения аналогична примеру, рассмотренному в разделе 5.1 для одиночных бинов, но теперь бин `CommentRepository` является прототипным. Взаимосвязи между бинами показаны на рис. 5.8.

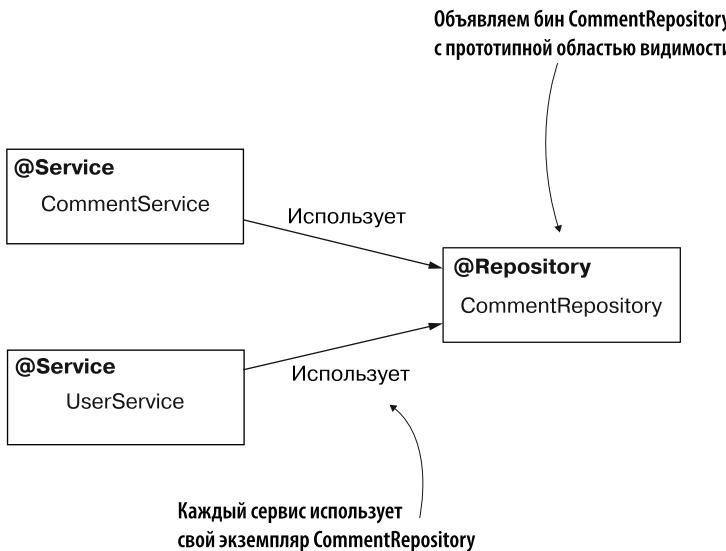


Рис. 5.8. Все классы сервиса делают запрос бина `CommentRepository`. Поскольку бин `CommentRepository` является прототипным, каждый сервис получает свой экземпляр `CommentRepository`

В следующем фрагменте кода показано определение класса `CommentRepository`. Обратите внимание, что аннотация `@Scope`, стоящая перед классом, изменяет область видимости бина на прототипную:

```
@Repository
@Scope(BeanDefinition.SCOPE_PROTOTYPE)
public class CommentRepository { }
```

Два класса сервисов запрашивают экземпляр типа `CommentRepository` посредством аннотации `@Autowired`. Класс `CommentService` выглядит так:

```
@Service
public class CommentService {

    @Autowired
    private CommentRepository commentRepository;

    public CommentRepository getCommentRepository() {
        return commentRepository;
    }
}
```

В этом фрагменте кода класс `UserService` также запрашивает экземпляр бина `CommentRepository`. Чтобы сообщить Spring, где находятся классы

со стереотипными аннотациями, в классе конфигурации нужно использовать аннотацию `@ComponentScan`:

```
@Configuration
@ComponentScan(basePackages = {"services", "repositories"})
public class ProjectConfig {
}
```

Теперь добавим в проект класс `Main` и проверим, как Spring будет внедрять бин `CommentRepository`. Класс `Main` показан в листинге 5.6.

Листинг 5.6. Проверка поведения Spring при внедрении прототипного бина в классе `Main`

```
public class Main {

    public static void main(String[] args) {
        var c = new AnnotationConfigApplicationContext(ProjectConfig.class);

        var s1 = c.getBean(CommentService.class); | Получить из контекста
        var s2 = c.getBean(UserService.class); | ссылки на бины сервисов

        boolean b = ←
            s1.getCommentRepository() == s2.getCommentRepository();

        System.out.println(b);   Сравнить ссылки для внедряемых экземпляров CommentRepository.
    }                           Поскольку CommentRepository является прототипным бином,
                                результатом сравнения всегда будет false
}
```

5.2.2. Практическое применение прототипных бинов

До сих пор мы рассматривали то, как Spring управляет прототипными бинами, обращая внимание на поведение фреймворка. Далее мы сфокусируемся на сценариях использования и рассмотрим, в каких случаях следует применять прототипные бины в реальных приложениях. Как и в случае с одиночными бинами из подраздела 5.1.2, мы учтем уже знакомые нам свойства прототипных бинов и проанализируем, когда такие бины могут быть эффективными, а когда их следует избегать (и использовать вместо них одиночные бины).

Прототипные бины будут вам встречаться не так часто, как одиночные. Но есть хорошее правило, которым можно пользоваться, решая, делать ли бин прототипным. Напомню, что одиночные бины не особенно хорошо подходят для изменяемых объектов. Предположим, что мы создаем объект `CommentProcessor`, который обрабатывает и валидирует комментарии. Некий сервис может использовать `CommentProcessor` для реализации сценария использования. Но в `CommentProcessor` в качестве атрибута хранится обрабатываемый комментарий, и методы этого объекта изменяют данный атрибут (рис. 5.9).

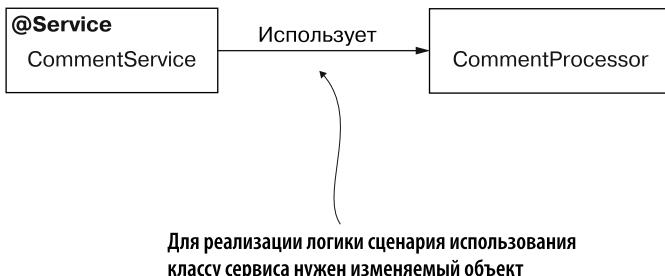


Рис. 5.9. Класс сервиса использует изменяемый объект для реализации логики сценария

Реализация бина CommentProcessor представлена в листинге 5.7.

Листинг 5.7. Изменяемый объект: возможный кандидат для прототипного бина

```
public class CommentProcessor {
    private Comment comment;

    public void setComment(Comment comment) {
        this.comment = comment;
    }

    public void getComment() {
        return this.comment;
    }

    public void processComment() { ←
        // изменение атрибута comment
    }

    public void validateComment() { ←
        // валидация и изменение атрибута comment
    }
}
```

Эти два метода изменяют значение атрибута Comment

В листинге 5.8 показан сервис, в котором посредством класса CommentProcessor реализуется сценарий использования. В методе сервиса создается экземпляр CommentProcessor с помощью конструктора класса CommentProcessor, а затем полученный экземпляр применяется в логике метода.

Листинг 5.8. Сервис, реализующий сценарий использования с помощью изменяемого объекта

```
@Service
public class CommentService {

    public void sendComment(Comment c) {
        CommentProcessor p = new CommentProcessor(); ←
        Создаем экземпляр CommentProcessor
    }
}
```

```

    p.setComment(c);
    p.processComment(c);
    p.validateComment(c);
    c = p.getComment(); ←
    // делаем что-нибудь еще
}
}                                | Используем экземпляр CommentProcessor,
                                    | чтобы изменить экземпляра Comment
                                    |
                                    | Получаем измененный
                                    | экземпляр Comment
                                    | и используем его
                                    | для чего-то еще

```

Объект `CommentProcessor` даже не является бином, размещенным в контексте Spring. А должен ли им быть? Исключительно важно задать себе этот вопрос, прежде чем сделать какой-либо объект бином. Помните, что в этом есть смысл, только если необходимо передать объект под управление Spring, чтобы расширить его возможности за счет функционала фреймворка. Для нашего сценария в том виде, в каком он существует сейчас, объект `CommentProcessor` не должен быть бином.

Но предположим, что бин `CommentProcessor` должен использовать объект `CommentRepository` для хранения каких-то данных и `CommentRepository` является бином, размещенным в контексте Spring (рис. 5.10).

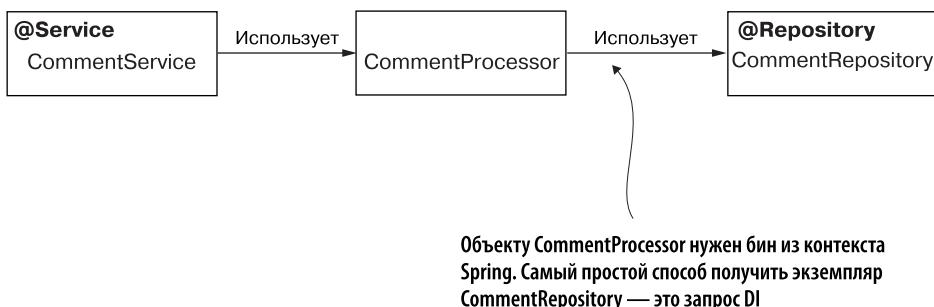


Рис. 5.10. Если объекту `CommentProcessor` нужен экземпляр `CommentRepository`, самый простой способ его получить — это запрос DI. Но для этого нужно сообщить Spring о существовании `CommentProcessor`, так что объект `CommentProcessor` должен быть бином в контексте фреймворка

В данном сценарии `CommentProcessor` должен быть бином, чтобы использовать предоставляемую Spring возможность DI. Как правило, если нужно расширить возможности объекта за счет функционала Spring, его следует сделать бином.

Итак, мы сделаем `CommentProcessor` бином и добавим его в контекст Spring. Но должен ли этот бин быть одиночным? Нет. Если объявить его одиночным и разные потоки начнут его использовать конкурентно, наступит состояние гонки (описанное в подразделе 5.1.2). Мы никогда не будем знать точно, какой именно комментарий, предоставленный потоком, обрабатывается в данный момент и правильно ли он был обработан. В данном сценарии нам нужно, чтобы каждый вызов метода получал свой экземпляр объекта `CommentProcessor`.

Для этого изменим класс `CommentProcessor` так, чтобы бин был прототипным, как показано в следующем фрагменте кода:

```
@Component
@Scope(BeanDefinition.SCOPE_PROTOTYPE)
public class CommentProcessor {

    @Autowired
    private CommentRepository commentRepository;

    // остальной код
}
```

Теперь мы можем получить из контекста Spring экземпляр `CommentProcessor`. Но будьте внимательны! Данный экземпляр будет вам нужен при каждом вызове метода `sendComment()`, поэтому запрос бина должен находиться внутри этого метода. Чтобы получить такой результат, нужно внедрить контекст Spring (`ApplicationContext`) непосредственно в бин `CommentService`, используя аннотацию `@Autowired`. В методе `sendComment()` мы получаем экземпляр `CommentProcessor`, вызывая `getBean()` для контекста приложения, как показано в листинге 5.9.

Листинг 5.9. Использование `CommentProcessor` как прототипного бина

```
@Service
public class CommentService {

    @Autowired
    private ApplicationContext context;

    public void sendComment(Comment c) {
        CommentProcessor p =
            context.getBean(CommentProcessor.class); ←
        p.setComment(c);
        p.processComment(c);
        p.validateComment(c);

        c = p.getComment();
        // сделать что-нибудь еще
    }
}
```

Здесь каждый раз
предоставляется новый
экземпляр `CommentProcessor`

Не следует внедрять `CommentProcessor` непосредственно в бин `CommentService` — это было бы ошибкой. Бин `CommentService` является одиночным, а следовательно, Spring создает только один экземпляр этого класса. Таким образом, при создании бина `CommentService` Spring сразу внедрит и все зависимости класса. В итоге получим единственный экземпляр `CommentProcessor`. И этот уникальный экземпляр будет использоваться при каждом вызове метода `sendComment()`, поэтому при наличии нескольких потоков возникнет такое же состояние гонки, как и в случае одиночного бина. Данная ситуация продемонстрирована

в следующем листинге. Вы можете использовать это упражнение, чтобы убедиться, что Spring ведет себя именно так.

Листинг 5.10. Внедрение прототипного бина в одиничный бин

```
@Service
public class CommentService {
    @Autowired
    private CommentProcessor p; ← Spring внедряет этот бин при создании бина
                                CommentService. Однако, поскольку CommentService
                                является одиничным бином, Spring также сразу
                                создаст и внедрит бин CommentProcessor

    public void sendComment(Comment c) {
        p.setComment(c);
        p.processComment(c);
        p.validateComment(c);

        c = p.getComment();
        // что-нибудь еще
    }
}
```

В завершение этого раздела я поделюсь своим мнением об использовании прототипных бинов. Как правило, в тех приложениях, которые я разрабатываю, я стараюсь по возможности не использовать ни прототипные бины, ни вообще изменяемые экземпляры. Но иногда возникает необходимость в рефакторинге или другой обработке старых приложений. Однажды мне встретился такой случай рефакторинга: нужно было внедрить Spring в старое приложение. В нем использовалось много изменяемых объектов, и быстро модифицировать все соответствующие фрагменты кода было невозможно. Пришлось создать прототипный бин — это позволило выполнить рефакторинг всех этих мест по очереди.

В качестве резюме сравним бины с одиничной и прототипной областями видимости. Их характеристика представлена в табл. 5.1.

Таблица 5.1. Краткое сравнение бинов с одиничной и прототипной областями видимости

Одиночные бины	Прототипные бины
1. Фреймворк ассоциирует имя с существующим экземпляром объекта	1. Имя ассоциируется с типом
2. При обращении к бину по имени каждый раз получаем один и тот же экземпляр объекта	2. При обращении к бину по имени каждый раз получаем новый экземпляр
3. Можно настроить Spring так, чтобы экземпляр создавался при загрузке контекста или при первом обращении к этому экземпляру	3. Фреймворк всегда создает экземпляры объектов с прототипной областью видимости при обращении к бину
4. По умолчанию Spring создает бины с одиничной областью видимости	4. Необходимо явно обозначить бин как прототипный
5. Не рекомендуется создавать одиничные бины с изменяемыми атрибутами	5. У прототипного бина могут быть изменяемые атрибуты

РЕЗЮМЕ

- В Spring область видимости бина определяет способ управления экземплярами объекта.
- В Spring существует две области видимости бинов: одиночная и прототипная.
 - В случае одиночной области видимости Spring управляет объектами непосредственно в контексте. У каждого экземпляра есть уникальное имя, которое всегда указывает на этот экземпляр. По умолчанию Spring создает бины с одиночной областью видимости.
 - В случае прототипной области видимости Spring учитывает только тип объекта. С каждым типом ассоциировано уникальное имя. При каждом обращении к бину по имени Spring создает новый экземпляр этого типа.
- Можно настроить Spring на создание одиночных бинов как при инициализации контекста (немедленное), так и при первом обращении к бину («левниковое»). По умолчанию все бины создаются немедленно.
- На практике обычно используются одиночные бины. Поскольку в этом случае при обращении по имени каждый раз возвращается один и тот же экземпляр, на данный экземпляр могут ссылаться несколько потоков. Поэтому рекомендуется делать такие экземпляры неизменяемыми. Но если вы решите выполнять операции, изменяющие атрибуты бина, необходимо обеспечить синхронизацию потоков.
- Если нужно сделать бином изменяемый объект, лучше использовать прототипную область видимости.
- Будьте внимательны при внедрении прототипных бинов в бины с одиночной областью видимости. Помните, что экземпляры одиночных бинов всегда используют один и тот же экземпляр прототипных бинов, которые Spring внедряет в момент создания одиночного экземпляра. Это порочная практика, поскольку обычно бин объявляют прототипным именно для того, чтобы при каждом обращении к нему получать другой экземпляр.

6

Аспекты и АОП в Spring

В этой главе

- ✓ Что такое аспектно-ориентированное программирование (АОП).
- ✓ Как использовать аспекты.
- ✓ Как строить цепочки выполнения аспектов.

До сих пор, рассматривая контекст Spring, мы использовали только одну функцию — DI, которая основана на принципе IoC. С помощью DI фреймворк управляет созданными объектами, их можно в любой момент запросить и применить. Как было показано в главах 2–5, чтобы получить ссылку на бин, обычно используется аннотация `@Autowired`. Когда некая функция запрашивает такой объект из контекста Spring, принято говорить, что Spring «внедряет» объект туда, откуда он был запрошен. В этой главе вы научитесь использовать еще одну эффективную технологию, основанную на принципе IoC, — *аспекты*.

Аспекты — это способ, которым фреймворк перехватывает вызов метода и может изменить его выполнение. Вы можете повлиять на метод при определенном его вызове. Данная технология позволяет извлечь часть логики, принадлежащей методу. Существуют сценарии, в которых после отделения части кода метод выглядит более понятным (рис. 6.1). Это позволяет разработчику, разбирающему логику метода, сконцентрировать внимание только на важных деталях. Далее будет показано, как реализовать аспекты и в каких случаях это следует делать. Аспекты — эффективное средство, но, как говорил дядя Питера Паркера, «чем больше сила, тем больше ответственность!» Бездумное использование

аспектов может закончиться плохо поддерживаемым приложением, то есть прямой противоположностью тому, чего мы добиваемся. Такой подход называется *аспектно-ориентированным программированием* (АОП).



Рис. 6.1. Некоторые части кода не стоит держать там же, где и бизнес-логику, — иначе бывает трудно понять, что делает приложение. Решение проблемы состоит в том, чтобы отделить часть кода от бизнес-логики посредством аспектов. В этом комиксе программист Джейн растеряна, увидев множество операций записи сообщений в журнал вперемешку с кодом бизнес-логики. Граф Дракула знакомит ее с магией аспектов, выделяя в аспект все записи в журнал

Еще одна существенная причина изучения аспектов состоит в том, что на их основе реализованы многие важнейшие функции Spring. Позже, когда вы

столкнетесь со специфичной проблемой, понимание принципов работы фреймворка сэкономит вам многие часы отладки. Яркий пример одной из главных возможностей Spring, где используются аспекты, — это *управление транзакциями*, о котором мы поговорим в главе 13. Данная функция нужна в большинстве современных приложений для обеспечения согласованности данных. Еще одна важная опция, основанная на аспектах, — это настройка системы безопасности, позволяющая защитить данные приложения и гарантировать, что эти сведения не смогут увидеть или изменить нежелательные лица. Дабы глубоко понимать, что происходит в приложении при использовании этих функций, прежде всего необходимо изучить аспекты.

Начнем с введения в теорию аспектов в разделе 6.1. Там вы познакомитесь с принципами работы аспектов. Изучив основы, в разделе 6.2 вы научитесь создавать аспекты. Мы разработаем пример для заданного сценария и будем его использовать, чтобы продемонстрировать на нем наиболее полезные синтаксические конструкции применения аспектов. В разделе 6.3 вы узнаете, что происходит, если есть несколько аспектов, перехватывающих один и тот же метод, и как следует поступать в таких случаях.

6.1. АСПЕКТЫ В SPRING

Рассмотрим принципы работы аспектов и основные понятия, с которыми вы столкнетесь при использовании аспектов. Когда вы научитесь их применять, у вас появится больше способов сделать приложение удобным в обслуживании. А еще вы поймете, каким образом некоторые функции Spring подключаются к приложениям. Затем, в разделе 6.2, мы перейдем непосредственно к реализации примера. Приступая к написанию кода, важно понимать, что именно хотите реализовать — только в этом случае аспекты будут эффективны.

Аспект — это просто часть логики, которую фреймворк реализует, вызывая определенные, выбранные вами методы. При разработке аспекта нужно определить следующее:

- *какой* код должен выполнять Spring при вызове данных методов. Именно это и называется *аспектом*;
- *когда* приложение должно выполнять логику аспекта (например, перед, после или вместо вызова метода). Это называется *советом*;
- выполнение *каких* методов должен перехватывать фреймворк и реализовывать аспект. Это называется *срезом*.

В терминологии аспектов вам также встретится понятие «*точка соединения*» — событие, запускающее выполнение аспекта. Но в Spring это событие всегда одно и то же — вызов метода.

Как при внедрении зависимости, при использовании аспектов объекты, к которым применяются аспекты, должны находиться под управлением фреймворка. Для добавления бинов в контекст Spring, чтобы фреймворк мог их контролировать и применить к ним созданные вами аспекты, вы будете использовать способы, с которыми познакомились в главе 2. Бин, в котором объявляется прерываемый аспектом метод, называют *целевым объектом*. Все эти новые термины сведены в одну схему на рис. 6.2.

Чтобы стать для аспекта целевым, объект
должен быть бином, размещенным в контексте Spring.
Spring должен иметь возможность
управлять этим объектом

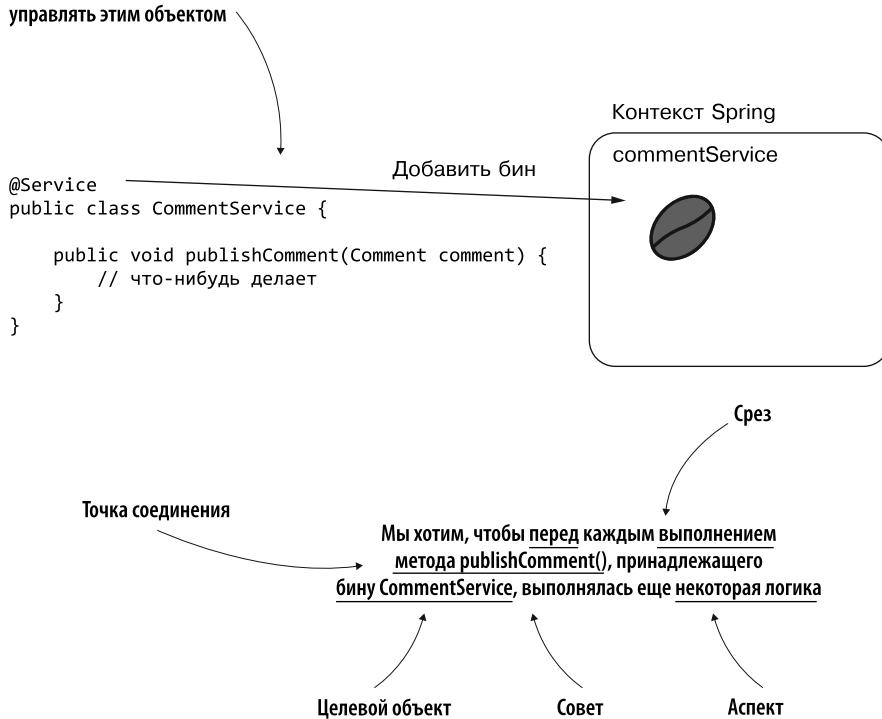


Рис. 6.2. Терминология аспектов. При вызове определенного метода (точки соединения) Spring выполняет некоторую логику (аспект). Нужно определить, в какой момент данная логика должна реализоваться относительно точки соединения (например, перед ней). Вот это «в какой момент» и называется советом. Чтобы фреймворк мог перехватить выполнение метода, объект, в котором определен метод, должен быть бином, размещенным в контексте Spring. Таким образом, этот бин становится целевым объектом для аспекта

Но каким образом Spring перехватывает выполнение метода и применяет логику аспекта? Как уже говорилось выше, для этого соответствующий объект должен

быть бином в контексте Spring. Но, поскольку мы сделали этот объект целевым для аспекта, Spring не даст нам прямую ссылку на бин, если мы запросим его из контекста. Вместо этого Spring предоставит объект, который вызывает не метод, а логику аспекта. Принято говорить, что взамен бину Spring предлагает *прокси-объект*. Теперь при получении бина из контекста нам всегда будет даваться его прокси-объект, независимо от того, вызывается ли непосредственно метод `getBean()` для контекста, или же используется DI (рис. 6.3). Данная технология называется *вплетением*.

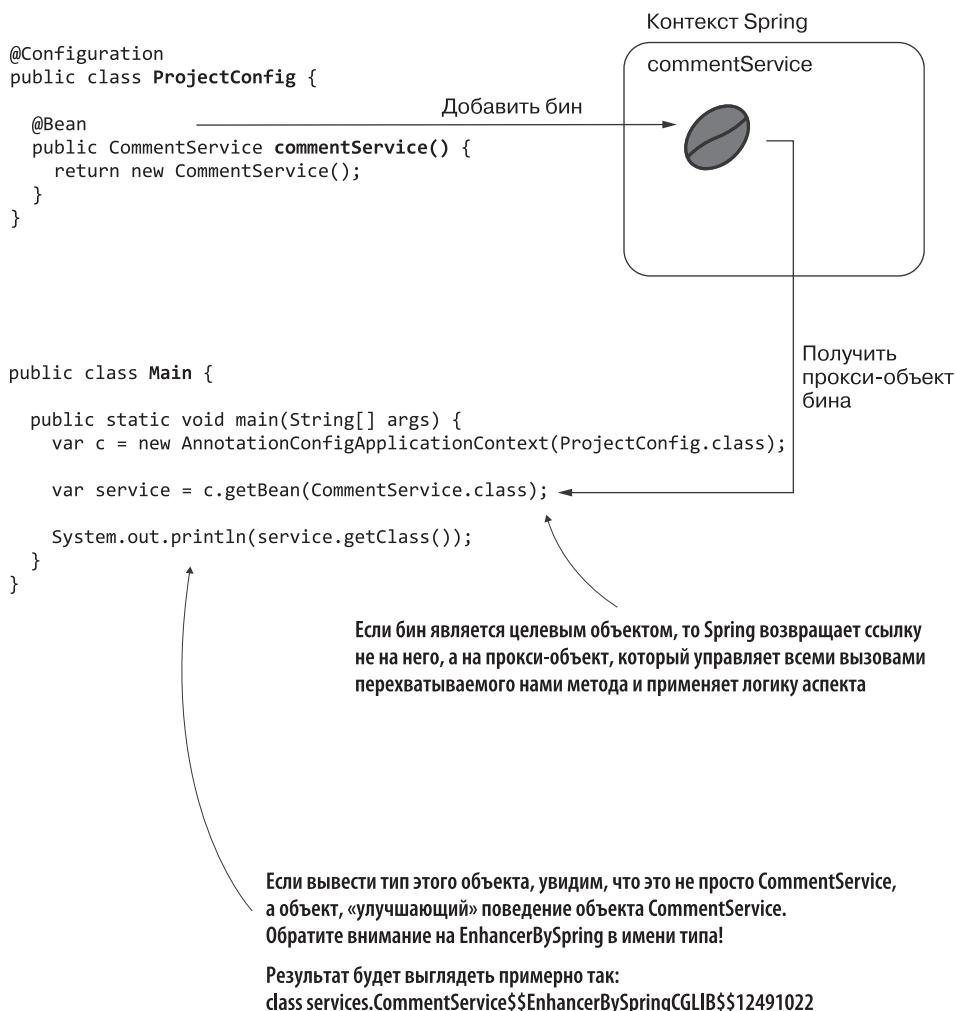
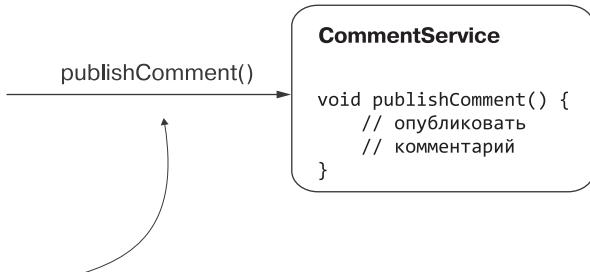


Рис. 6.3. Вплетение аспекта. Spring предоставляет ссылку не на сам бин, а на прокси-объект, который перехватывает вызовы метода и управляет логикой аспекта

На рис. 6.4 показано сравнение процессов, когда вызов метода не перехватывается и перехватывается аспектом. Как видим, в случае с аспектом предполагается, что метод вызывается через прокси-объект, предоставленный Spring. Прокси же применяет логику аспекта и делегирует вызов самому методу.

Без аспекта



Если метод `publishComment()` не перехватывается аспектом, то при вызове этого метода сразу выполняется логика, описанная в классе `CommentService`

С аспектом



Если для метода определен аспект, то вызов метода осуществляется через прокси-объект, созданный Spring. Этот прокси-объект применяет логику аспекта и затем делегирует вызов исходному методу

Рис. 6.4. Если у метода нет аспекта, то вызов передается непосредственно самому методу. Если для метода определен аспект, то вызов проходит через прокси-объект. Этот прокси-объект применяет логику, описанную в аспекте, и затем делегирует вызов исходному методу

Теперь, когда вы получили общее представление о том, что такое аспекты и как они работают в Spring, пойдем дальше и рассмотрим синтаксические конструкции, необходимые для реализации аспектов. В разделе 6.2 мы опишем, а затем выполним конкретный сценарий с помощью аспектов.

6.2. РЕАЛИЗАЦИЯ АСПЕКТОВ В SPRING С ПОМОЩЬЮ АОП

Далее вы познакомитесь с наиболее актуальными синтаксическими конструкциями аспектов, используемыми в реальных приложениях. Мы рассмотрим сценарий и реализуем его требования с помощью аспектов. К концу этого раздела вы научитесь применять синтаксис аспектов для решения наиболее типичных практических задач.

Предположим, что есть некое приложение с классами сервисов, реализующими многочисленные сценарии использования. В соответствии с новыми требованиями приложение должно запоминать время начала и конца выполнения каждого сценария. На совещании вы предложили взять на себя разработку функционала по записи в журнал всех событий старта и окончания сценариев использования.

В подразделе 6.2.1 мы решим эту задачу простейшим из возможных способов с использованием аспекта. Так вы поймете, что нужно для внедрения данного модуля. Далее я буду постепенно раскрывать новые подробности применения аспектов. В подразделе 6.2.2 будет показано, как аспект использует и даже изменяет параметры перехватываемого метода или возвращаемое этим методом значение. В подразделе 6.2.3 вы научитесь отмечать аннотациями методы, нужные вам для специфических задач. Разработчики часто используют аннотации, чтобы отмечать методы, которые должны перехватываться аспектами. Как вы поймете из следующих глав, аннотации применяются для многих функций Spring. В подразделе 6.2.4 вы познакомитесь еще с несколькими аннотациями советов, которые можно использовать для аспектов Spring.

6.2.1. Реализация простого аспекта

Рассмотрим решение нашей задачи с помощью простого аспекта. Создадим новый проект и определим класс сервиса. В этом классе создадим метод, который будем использовать, чтобы проверить наше решение и убедиться, что определенный нами аспект в итоге работает так, как нужно.

Данный пример находится в проекте sq-ch6-ex1. Кроме зависимости от контекста Spring, в этот раз нам понадобится зависимость от аспектов Spring. Не забудьте добавить их в файл pom.xml:

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.8.RELEASE</version>
</dependency>
<dependency> ←———— Эта зависимость нужна для реализации аспектов
    <groupId>org.springframework</groupId>
```

```
<artifactId>spring-aspects</artifactId>
<version>5.2.8.RELEASE</version>
</dependency>
```

Чтобы пример был короче и вы могли сконцентрировать внимание на синтаксисе аспектов, мы будем рассматривать только один сервисный объект `CommentService` и его сценарий использования `publishComment(Comment comment)`. Этот метод, определенный в классе `CommentService`, принимает параметр типа `Comment`. `Comment` — это класс модели, который выглядит так:

```
public class Comment {
    private String text;
    private String author;

    // геттеры и сеттеры
}
```

ПРИМЕЧАНИЕ

Как вы помните из главы 4, класс модели — это класс, который моделирует данные, обрабатываемые приложением. В нашем случае класс `Comment` описывает комментарий и его атрибуты: `text` и `author`. Класс сервиса реализует сценарии использования приложения. В главе 4 были описаны и другие обязанности, которые мы применяли в примерах.

В листинге 6.1 содержится определение класса `CommentService`. Мы снабдили класс стереотипной аннотацией `@Service`, чтобы создать бин в контексте Spring. В `CommentService` определен метод `publishComment(Comment comment)`, который реализует заданный сценарий использования.

Вы также заметите, что в данном примере для записи сообщений в консоль я использовал не `System.out`, а объект типа `Logger`. В реальных приложениях для выполнения данного действия `System.out` не применяется. Вместо этого обычно используют фреймворки журналирования, которые предоставляют больше возможностей для записи в журнал и стандартизации журнальных сообщений. Вот несколько хороших:

- Log4j (<https://logging.apache.org/log4j/2.x/>);
- Logback (<http://logback.qos.ch/>);
- Java Logging API, в комплект которого входит JDK (<http://mng.bz/v4Xq>).

Фреймворки журналирования совместимы со всеми Java-приложениями, независимо от того, используется ли в них Spring. Я не упоминал их в предыдущих примерах, чтобы вас не отвлекать. Но теперь вы достаточно хорошо изучили Spring, чтобы применять эти фреймворки в наших примерах и чтобы ближе познакомиться с синтаксическими конструкциями, типичными для реальных приложений.

Листинг 6.1. Класс CommentService, используемый в примерах

```
@Service ← Мы использовали здесь стереотипную аннотацию, чтобы создать бин в контексте Spring
public class CommentService {
    private Logger logger = ←
        Logger.getLogger(CommentService.class.getName());
    public void publishComment(Comment comment) {
        logger.info("Publishing comment:" + comment.getText()); ←
    }
}
```

Мы внедрили объект logger, чтобы каждый раз, когда выполняется сценарий использования, выводилось сообщение в консоль приложения

В этом методе определяется сценарий использования для нашей демонстрации

В данном примере я использовал средства журналирования JDK, чтобы не добавлять в проект лишних зависимостей. При объявлении объекта logger необходимо дать ему имя в качестве параметра. Затем данное имя будет появляться в записях журнала, и вам будет проще отслеживать источники сообщений. Для этого часто используют имя класса; я тоже так поступил в нашем примере: `CommentService.class.getName()`.

Нам также нужно создать класс конфигурации, чтобы сообщить Spring, где находятся классы со стереотипными аннотациями. Я разместил класс сервиса в пакете `services`, и поэтому мне нужно добавить в класс конфигурации аннотацию `@ComponentScan`:

```
@Configuration
@ComponentScan(basePackages = "services") ←
public class ProjectConfig {
```

С помощью аннотации `@ComponentScan` мы сообщаем Spring, где находятся классы со стереотипными аннотациями

Напишем класс `Main`, который вызывает метод `publishComment()`, определенный в классе сервиса, и проследим за поведением приложения. Класс `Main` представлен в листинге 6.2.

Листинг 6.2. Класс Main, с помощью которого мы проверим поведение приложения

```
public class Main {
    public static void main(String[] args) {
        var c = new AnnotationConfigApplicationContext(ProjectConfig.class);
        var service = c.getBean(CommentService.class); ←
        Comment comment = new Comment(); ←
        comment.setText("Demo comment");
        comment.setAuthor("Natasha"); ←
        service.publishComment(comment); ←
    }
}
```

Получить из контекста бин CommentService

Создать экземпляр Comment и передать его как параметр в метод publishComment()

Вызвать метод publishComment()

Запустив приложение, вы увидите в консоли примерно следующее:

```
Sep 26, 2020 12:39:53 PM services.CommentService publishComment
INFO: Publishing comment:Demo comment
```

Эти сообщения генерируются методом `publishComment()`. Так работает приложение перед решением описанной ранее задачи. Напомню: нам нужно выводить в консоль сообщения перед вызовом метода сервиса и после него. Теперь добавим в проект класс аспекта, который будет перехватывать вызов метода и добавлять вывод сообщений до и после него.

Чтобы создать аспект, нужно выполнить следующие действия (рис. 6.5).

1. Активируйте механизм аспектов в Spring-приложении, поставив перед классом конфигурации аннотацию `@EnableAspectJAutoProxy`.
2. Создайте новый класс и внедрите перед ним аннотацию `@Aspect`. Затем, используя либо аннотацию `@Bean`, либо одну из стереотипных аннотаций, добавьте бин этого класса в контекст Spring.
3. Определите метод, в котором будет реализована логика аспекта, и с помощью аннотаций советов сообщите Spring, где и какие методы следует перехватывать.
4. Реализуйте логику аспекта.

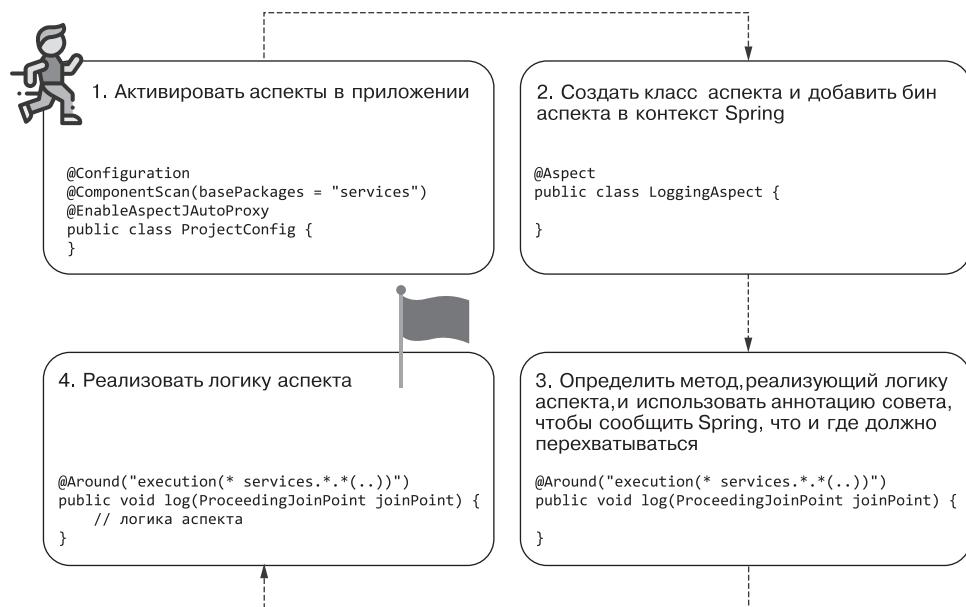


Рис. 6.5. Чтобы реализовать аспект, нужно выполнить четыре простые операции. Вначале активировать использование аспектов в приложении. Затем создать класс аспекта, определить в нем метод и сообщить Spring, что и где следует перехватывать. И в конце реализовать логику аспекта

Шаг 1. Активировать механизм аспектов в приложении

Прежде всего необходимо сообщить Spring, что в приложении используются аспекты. Когда вы применяете один из механизмов фреймворка, следует его ясно внедрить, снабдив класс конфигурации соответствующей аннотацией. Как правило, имена этих аннотаций начинаются с `Enable`. Далее в книге вы познакомитесь еще с несколькими аннотациями, активирующими различные функции Spring. В данном примере, чтобы реализовать возможность использования аспектов, нам нужна аннотация `@EnableAspectJAutoProxy`. Класс конфигурации должен выглядеть так, как показано в листинге 6.3.

Листинг 6.3. Активация механизма аспектов в Spring-приложении

```
@Configuration
@ComponentScan(basePackages = "services")
@EnableAspectJAutoProxy ← Активирует аспекты в Spring-приложении
public class ProjectConfig {  
}
```

Шаг 2. Создать класс с определением аспекта и добавить экземпляр этого класса в контекст Spring

Теперь нужно создать в контексте Spring новый бин, в котором будет определен аспект. В данном объекте находятся методы, которые будут перехватывать вызов заданного метода и добавлять туда свою логику. Определение нового класса представлено в листинге 6.4.

Листинг 6.4. Определение класса аспекта

```
@Aspect
public class LoggingAspect {
    public void log() {
        // позже мы его реализуем
    }
}
```

Чтобы добавить этот экземпляр в контекст Spring, вы можете воспользоваться любым способом, описанным в главе 2. Если выберете аннотацию `@Bean`, не забудьте изменить класс конфигурации, как показано в следующем фрагменте кода. Разумеется, вы также можете воспользоваться стереотипными аннотациями:

```
@Configuration
@ComponentScan(basePackages = "services")
@EnableAspectJAutoProxy
public class ProjectConfig {  
  
    @Bean ← Добавляет экземпляр класса LoggingAspect в контекст Spring
    public LoggingAspect aspect() {
        return new LoggingAspect();
    }
}
```

Напомню: данный объект необходимо сделать бином и добавить в контекст Spring, чтобы передать под контроль фреймворка. Именно поэтому я уделил так много внимания способам управления контекстом Spring в главах 2–5. Вы будете постоянно использовать эти навыки при разработке Spring-приложений.

Учтите, что аннотация `@Aspect` не является стереотипной. С помощью `@Aspect` мы сообщаем Spring, что в данном классе содержится определение аспекта, но это не значит, что Spring сразу создаст для него бин. Для выполнения последнего действия необходимо ясно прописать одну из синтаксических конструкций, о которых вы узнали в главе 2. Не добавить класс с аннотацией `@Aspect` в контекст — типичная ошибка. Мне часто приходилось видеть, как люди испытывали замешательство, забыв об этой особенности.

Шаг 3. С помощью аннотации совета сообщить Spring, какой метод и в какой момент нужно перехватывать

Теперь, когда определен класс аспекта, можно выбрать совет и снабдить метод соответствующей аннотацией. В листинге 6.5 показан метод с аннотацией `@Around`.

Листинг 6.5. Вплетение аспекта в выбранный метод с помощью аннотации совета

```
@Aspect
public class LoggingAspect {

    @Around("execution(* services.*.*(..))") ← Определение перехватываемых методов
    public void log(ProceedingJoinPoint joinPoint) {
        joinPoint.proceed(); ← Делегирование управления перехваченному методу
    }
}
```

Кроме аннотации `@Around`, вы также заметите здесь необычное строковое выражение в качестве значения аннотации и параметр, добавленный в метод аспекта. Что это такое?

Рассмотрим данные детали по очереди. Странный оборот, использованный в качестве параметра аннотации `@Around`, сообщает Spring, какой метод нужно перехватить. Пусть он вас не пугает! Этот язык выражений называется языком срезов AspectJ, но ничего не придется заучивать наизусть, чтобы им пользоваться. На практике сложные выражения вам не встретятся. Когда мне нужно написать что-то подобное, я обычно обращаюсь к документации (<http://mng.bz/4K9g>).

Теоретически на AspectJ можно составлять очень сложные выражения срезов, чтобы описывать определенные наборы вызовов методов, которые нужно перехватывать. Это действительно очень мощный язык. Но, как вы увидите далее, всегда лучше по возможности избегать сложных конструкций. В большинстве случаев можно найти варианты попроще.

Взгляните на использованное мной выражение (рис. 6.6). Оно означает, что Spring будет перехватывать любой метод, принадлежащий классу из пакета `services`, независимо от того, какой тип данных возвращает этот метод, к какому именно классу принадлежит, какое ему присвоено имя и какие параметры он принимает.



Рис. 6.6. Использованное в примере выражение среза AspectJ сообщает Spring, что нужно перехватывать вызовы всех методов из пакета `services`, независимо от типа возвращаемого значения, класса, к которому метод принадлежит, имени метода или принимаемых им параметров

При ближайшем рассмотрении данная конструкция не выглядит такой уж сложной, правда? Знаю, выражения срезов на AspectJ часто пугают новичков, но поверьте, вам не нужно становиться экспертом по AspectJ, чтобы использовать их в приложениях Spring.

Теперь обратим внимание на второй элемент, который я добавил в метод, — параметр `ProceedingJoinPoint`. Этот параметр обозначает перехватываемый метод. Главное, что он делает, — сообщает аспекту, в какой момент следует делегировать управление исходному методу.

Шаг 4. Реализация логики аспекта

В листинге 6.6 я добавил логику аспекта. Теперь аспект делает следующее.

1. Перехватывает метод.
2. Перед вызовом перехваченного метода выводит что-нибудь в консоль.
3. Вызывает перехваченный метод.
4. После вызова перехваченного метода выводит что-нибудь в консоль.

Поведение аспекта визуально представлено на рис. 6.7.

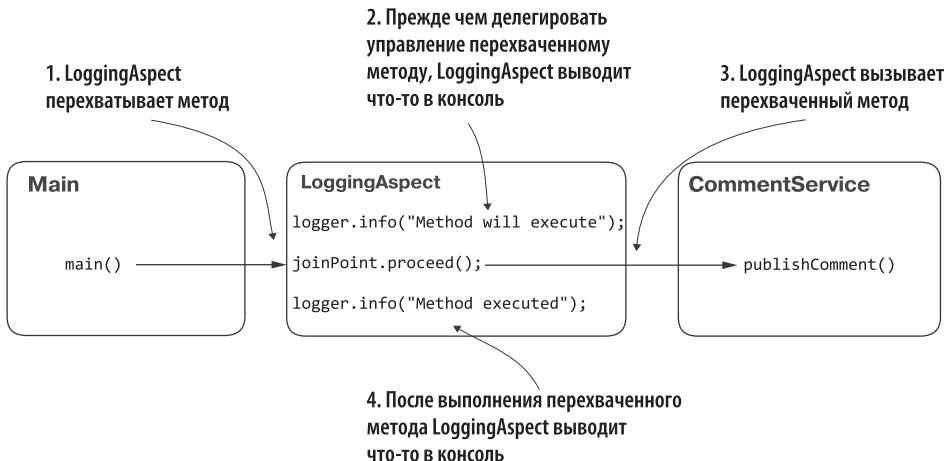


Рис. 6.7. Поведение аспекта. LoggingAspect служит оберткой для метода, выводя что-то перед вызовом этого метода и после него. Таким образом получаем простую реализацию аспекта

Листинг 6.6. Реализация логики аспекта

```

@Aspect
public class LoggingAspect {

    private Logger logger = Logger.getLogger(LoggingAspect.class.getName());

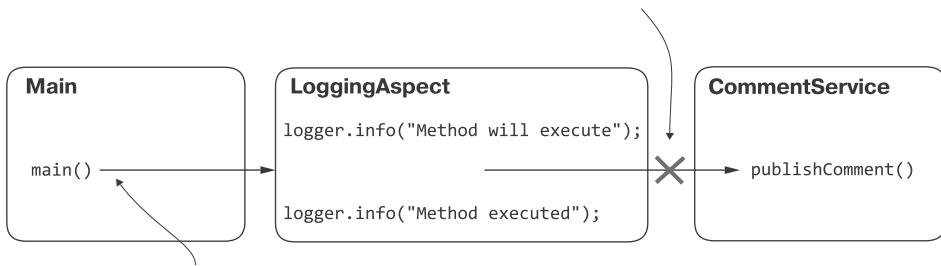
    @Around("execution(* services.*.*(..))")
    public void log(ProceedingJoinPoint joinPoint) throws Throwable {
        logger.info("Method will execute");           | Выводит сообщение в консоль перед
        joinPoint.proceed();                         | выполнением перехваченного метода
        logger.info("Method executed");             | Выводит сообщение в консоль после
                                                | выполнения перехваченного метода
    }
}

```

Метод `proceed()` параметра `ProceedingJoinPoint` вызывает перехваченный метод `publishComment()`, принадлежащий бину `CommentService`. Если не использовать `proceed()`, аспект никогда не будет делегировать управление перехваченному методу (рис. 6.8).

Мы даже можем реализовать логику, при которой перехваченный метод вообще не будет вызываться. Например, можно создать аспект, который будет проверять некие условия авторизации и затем решать, передавать ли управление дальше защищенному методу. Если условия авторизации не выполнены, аспект не будет делегировать контроль перехваченному методу, таким образом защищая его (рис. 6.9).

Если не вызвать метод proceed(), принадлежащий параметру ProceedingJoinPoint, то аспект никогда не делегирует управление перехваченному методу



Аспект выполняет свою логику и сразу возвращает управление методу main(). С точки зрения main() метод publishComment() выполнен

Рис. 6.8. Если не вызвать метод proceed() для параметра аспекта ProceedingJoinPoint, аспект не будет делегировать управление перехваченному методу; вместо этого просто выполнится аспект. Там, откуда вызывается перехваченный метод, не будут знать, что он на самом деле не был вызван

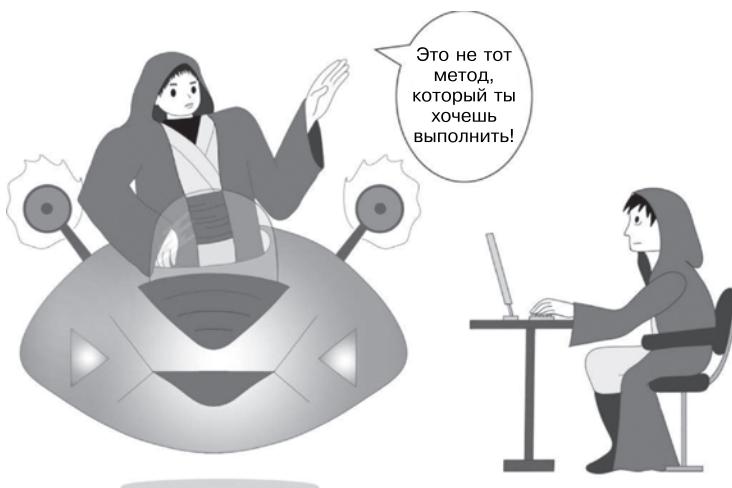


Рис. 6.9. Аспект может принять решение вообще не передавать управление перехваченному методу. Снаружи это выглядит, будто аспект дает тому, кто вызвал метод, ментальный приказ. В итоге выполняется не та логика, которую вызывающий объект ждал

Обратите также внимание, что метод proceed() выбрасывает исключение Throwable. proceed() рассчитан на то, чтобы выдавать любое исключение, поступающее от перехваченного метода. В данном примере я выбрал простейший

способ передать его дальше, но при необходимости можно обработать его с помощью блока `try-catch-finally`.

Вернемся к нашему приложению (проект `sq-ch6-ex1`). Мы обнаружим в консоли записи, выведенные и аспектом, и перехваченным методом. Сообщения должны выглядеть примерно так:

```
Sep 27, 2020 1:11:11 PM aspects.LoggingAspect log
INFO: Method will execute ← Эту строку выводит аспект
Sep 27, 2020 1:11:11 PM services.CommentService publishComment
INFO: Publishing comment:Demo comment ← Эту строку выводит перехваченный метод
Sep 27, 2020 1:11:11 PM aspects.LoggingAspect log
INFO: Method executed ← Эту строку выводит аспект
```

6.2.2. Изменение параметров и возвращаемого значения перехваченного метода

Как я уже говорил, аспекты невероятно эффективны. Они не только перехватывают метод и изменяют его выполнение, но также могут перехватывать параметры, использованные при вызове метода, и при необходимости модифицировать их или значение, возвращаемое методом. Далее мы перепишем пример, с которым до сих пор работали, чтобы показать, каким образом аспект может влиять на параметры перехваченного метода и возвращаемое им значение. Научившись это делать, вы еще больше расширите свои возможности в реализации аспектов.

Предположим, что нам нужно записывать в журнал параметры, используемые при вызове метода сервиса, а также то, что этот метод возвращает. Чтобы продемонстрировать реализацию подобного сценария, я вынес пример в отдельный проект `sq-ch6-ex2`. Поскольку нам также понадобится ссылка на то, что возвращает метод, я добавил в сервисный метод возвращаемое значение:

```
@Service
public class CommentService {

    private Logger logger = Logger.getLogger(CommentService.class.getName());

    public String publishComment(Comment comment) {
        logger.info("Publishing comment:" + comment.getText());
        return "SUCCESS"; ← В демонстрационных целях теперь этот метод возвращает значение
    }
}
```

Аспект легко получает имя перехватываемого метода и его параметры. Напомню, что любую информацию о перехватываемом методе (параметры, имя, целевой объект и т. п.) можно получить с помощью параметра `ProceedingJoinPoint`,

которым данный метод представлен в методе аспекта. В следующем фрагменте кода показано, как получить имя метода и параметры, использованные при его вызове, прежде чем перехватить вызов:

```
String methodName = joinPoint.getSignature().getName();
Object [] arguments = joinPoint.getArgs();
```

Теперь можно изменить аспект так, чтобы выводить данную информацию в консоль. Нужные изменения в методе аспекта показаны в листинге 6.7.

Листинг 6.7. Получение имени и параметров перехватываемого метода логикой аспекта

```
@Aspect
public class LoggingAspect {

    private Logger logger = Logger.getLogger(LoggingAspect.class.getName());

    @Around("execution(* services.*.*(..))")
    public Object log(ProceedingJoinPoint joinPoint) throws Throwable {
        String methodName =
            joinPoint.getSignature().getName();
        Object [] arguments = joinPoint.getArgs(); | Получение имени и параметров
                                                       | перехватываемого метода
        logger.info("Method " + methodName + | Вывод имени и параметров
                                                       | перехватываемого метода
            " with parameters " + Arrays.asList(arguments) +
            " will execute");
        Object returnedByMethod = joinPoint.proceed(); | Вызов перехватываемого
                                                       | метода
        logger.info("Method executed and returned " + returnedByMethod);

        return returnedByMethod; | Возвращение значения, которое
                           | возвращает перехватываемый метод
    }
}
```

Благодаря рис. 6.10 вам будет проще представить себе весь процесс. Обратите внимание, что аспект перехватывает вызов и получает доступ к параметрам метода и возвращаемому им значению.

Как показано в листинге 6.8, я изменил метод `main()` таким образом, чтобы выводилось значение, возвращаемое `publishComment()`.

Листинг 6.8. Вывод возвращаемого значения для контроля поведения аспекта

```
public class Main {

    private static Logger logger = Logger.getLogger(Main.class.getName());

    public static void main(String[] args) {
        var c = new AnnotationConfigApplicationContext(ProjectConfig.class);
```

```

var service = c.getBean(CommentService.class);

Comment comment = new Comment();
comment.setText("Demo comment");
comment.setAuthor("Natasha");

String value = service.publishComment(comment);

logger.info(value); ← Вывести значение, возвращаемое методом publishComment()
}

}

Метод main() вызывает метод publishComment()
для бина CommentService, но аспект
перехватывает этот вызов
}

```

Аспект выводит информацию о вызове,
а также параметры перехваченного метода
и возвращаемое им значение

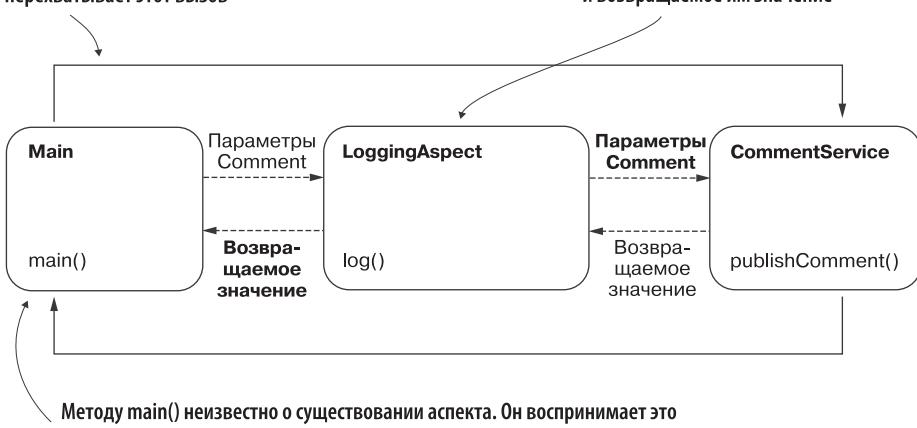


Рис. 6.10. Поскольку аспект перехватывает вызов метода, он может получить доступ к параметрам этого метода и значению, которое возвращается после выполнения метода. Со стороны метода `main()` это выглядит как прямой вызов `publishComment()` для бина `CommentService`. Вызывающий метод не знает, что вызов перехватывается аспектом

При выполнении приложения вы увидите в консоли значения, полученные методом `main()` от аспекта и перехваченного метода:

```

Sep 28, 2020 10:49:39 AM aspects.LoggingAspect log      Эти параметры выводит аспект
INFO: Method publishComment with parameters [Comment{text='Demo comment',
➥ author='Natasha'}] will execute
Sep 28, 2020 10:49:39 AM services.CommentService publishComment
INFO: Publishing comment:Demo comment ← Это сообщение выводит перехваченный метод
Sep 28, 2020 10:49:39 AM aspects.LoggingAspect log
INFO: Method executed and returned SUCCESS ←
Sep 28, 2020 10:49:39 AM main.Main main
INFO: SUCCESS ← Это значение, возвращаемое перехваченным методом, выводит метод main()

```

Но и на этом возможности аспектов не заканчиваются. Аспекты позволяют влиять на выполнение перехватываемого метода следующими способами:

- модифицируя значения параметров, передаваемых методу;
- изменяя значение, возвращаемое перехваченным методом и передаваемое вызывающему методу;
- выбрасывая исключение и передавая его вызывающему методу либо перехватывая и обрабатывая исключение, которое выдает перехваченный метод.

Есть очень много вариантов воздействия на перехватываемый метод. Вы можете даже полностью изменить его поведение (рис. 6.11). Но будьте осторожны! Меняя логику посредством аспекта, вы делаете часть этой логики прозрачной. Убедитесь, что не скрываете неочевидные моменты. Вообще, цель отделения части логики состоит в том, чтобы избежать дублирования кода и скрыть то, что не имеет отношения к делу, чтобы разработчику было легче сконцентрироваться на бизнес-логике. Собираясь создать аспект, поставьте себя на место того, кому понадобится разобраться в коде и быстро понять, что происходит с приложением.

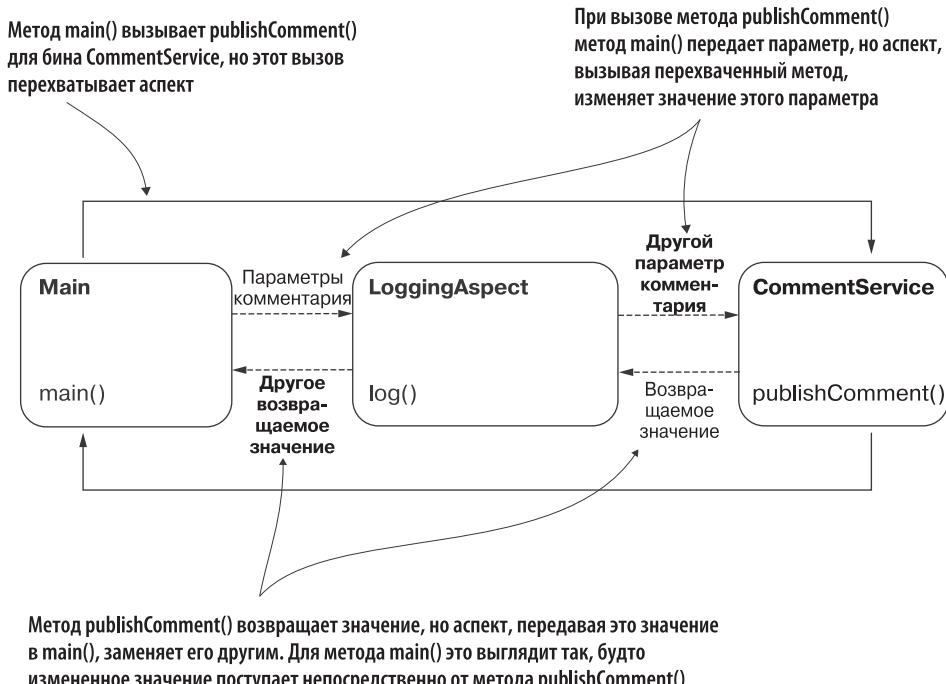


Рис. 6.11. Аспект может изменять параметры, использованные при вызове перехватываемого метода, а также возвращаемое значение, передаваемое от перехватываемого метода тому методу, который его вызвал. Это очень эффективная технология, которая дает возможность гибко контролировать перехватываемый метод

Проект sq-ch6-ex3 демонстрирует, каким образом аспекты могут влиять на вызов перехватываемого метода, изменяя его параметры или возвращаемое значение. В листинге 6.9 показано, что если вызвать метод `proceed()`, не передав ему никаких параметров, то аспект даст перехватываемому методу свои параметры. Но если передать методу `proceed()` параметр, представляющий собой массив объектов, то аспект даст перехватываемому методу этот массив вместо своих параметров. Аспект выводит в консоль значение, возвращаемое перехваченным методом, но вызывающий метод получает другое значение.

Листинг 6.9. Изменение параметров и возвращаемого значения

```
@Aspect
public class LoggingAspect {

    private Logger logger =
        Logger.getLogger(LoggingAspect.class.getName());

    @Around("execution(* services.*.*(..))")
    public Object log(ProceedingJoinPoint joinPoint) throws Throwable {
        String methodName = joinPoint.getSignature().getName();
        Object [] arguments = joinPoint.getArgs();

        logger.info("Method " + methodName +
            " with parameters " + Arrays.asList(arguments) +
            " will execute");

        Comment comment = new Comment();
        comment.setText("Some other text!");
        Object [] newArguments = {comment};

        Object returnedByMethod = joinPoint.proceed(newArguments); ←

        logger.info("Method executed and returned " + returnedByMethod);

        return "FAILED"; ←
    }
}
```

В качестве параметра
методу передается другой
экземпляр Comment

Значение, возвращаемое перехваченным методом, выводится
в консоль, а вызывающий метод получает другое значение

При запуске приложения получим результат, показанный ниже. Значения параметров, полученных методом `publishComment()`, отличаются от тех, которые были ему переданы вызывающим методом. Метод `publishComment()` возвращает одно значение, а `main()` получает другое:

<pre>Sep 29, 2020 10:43:51 AM aspects.LoggingAspect log INFO: Method publishComment with parameters [Comment{text='Demo comment', ➥ author='Natasha'}] will execute ←</pre>	<p style="text-align: right;">При вызове метода <code>publishComment()</code> ему передан комментарий с текстом Demo comment</p>
<pre>Sep 29, 2020 10:43:51 AM services.CommentService publishComment INFO: Publishing comment:Some other text! ←</pre>	<p style="text-align: right;">Метод <code>publishComment()</code> получает комментарий с текстом Some other text!</p>
<pre>Sep 29, 2020 10:43:51 AM aspects.LoggingAspect log INFO: Method executed and returned SUCCESS ←</pre>	<p style="text-align: right;">Метод <code>publishComment()</code> возвращает текст SUCCESS</p>
<pre>Sep 29, 2020 10:43:51 AM main.Main main INFO: FAILED ←</pre>	<p style="text-align: right;">Метод <code>main()</code> получает от <code>publishComment()</code> значение FAILED</p>

ПРИМЕЧАНИЕ

Я понимаю, что повторяюсь, но это действительно важный момент. Будьте осторожны при использовании аспектов! Применяйте их, только чтобы скрыть тот код, который не имеет отношения к делу и при этом совершенно понятен. Аспекты — очень мощная вещь, и они способны переманить вас на «темную сторону». Однажды вы уберете важные строки — и приложение станет трудно поддерживать. Будьте осторожны с аспектами!

Но нужен ли нам аспект, изменяющий параметры перехватываемого метода или возвращаемое этим методом значение? Да, иногда такие аспекты бывают полезны. Я описал все эти варианты, поскольку в следующих главах мы будем использовать некоторые возможности Spring, основанные на аспектах. Например, в главе 13 будут рассмотрены транзакции. Они в Spring как раз основаны на аспектах. Поэтому, когда мы дойдем до этой темы, вы поймете, что аспекты очень полезны.

Разобравшись в принципах работы аспектов, впоследствии вы будете значительно лучше понимать, как работает Spring в целом. Мне часто встречаются разработчики, которые начинают использовать фреймворк, не осознавая, на чем основан нужный им функционал. Неудивительно, что они часто вносят ошибки и уязвимости в приложения или же их продукты менее производительны и удобны, чем могли бы быть. Поэтому я советую всегда сначала изучить принципы работы того или иного инструмента и только потом его использовать.

6.2.3. Перехват методов с аннотациями

Далее мы рассмотрим важную методику, часто применяемую в Spring-приложениях, чтобы отметить методы, которые должны перехватываться аспектами, — использование аннотаций. Вы обратили внимание, сколько аннотаций мы уже знаем из примеров? Они так удобны, что с момента своего появления в Java 5 практически стали стандартом для конфигурации приложений, использующих те или иные фреймворки. Пожалуй, сейчас не найдется Java-фреймворка, в котором бы не было аннотаций. С их помощью также можно отмечать методы, которые должны перехватываться аспектами. Этот удобный синтаксис позволяет избежать написания сложных выражений среза AspectJ.

Для изучения данной методики мы создадим новый пример, подобный описанному ранее в этой главе. Мы добавим в класс `CommentService` три метода: `publishComment()`, `deleteComment()` и `editComment()`. Пример находится в проекте

sq-ch6-ex4. Мы хотим создать свою аннотацию и записывать в журнал данные о выполнении только тех методов, которые ею отмечены. Для этого нужно сделать следующее.

1. Определить аннотацию и сделать ее доступной при выполнении приложения. Мы назовем ее `@ToLog`.
2. С помощью еще одного выражения среза AspectJ сообщить методу аспекта, что нужно перехватывать методы с новой аннотацией.

Данные действия визуально представлены на рис. 6.12.

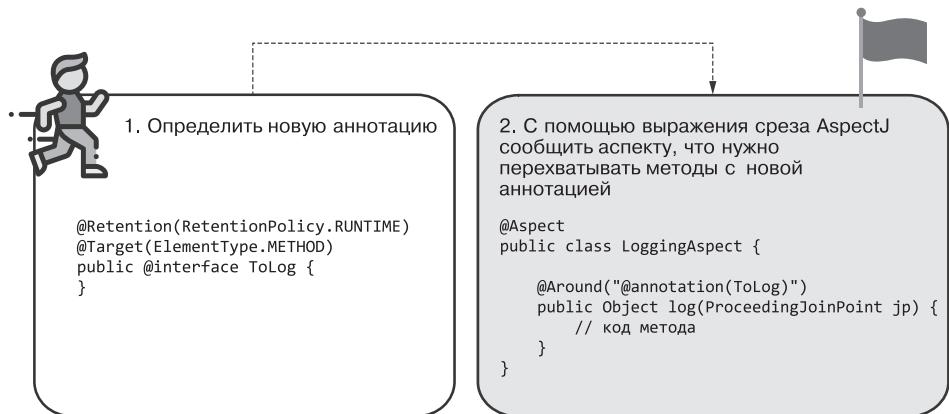


Рис. 6.12. Действия, необходимые для перехвата методов с аннотациями. Нужно создать новую аннотацию, которая будет сопровождать методы, нуждающиеся в перехвате, а затем с помощью выражения среза AspectJ настроить аспект на перехват методов с новой аннотацией

Логику аспекта менять не нужно. В этом примере аспект делает то же самое, что и в предыдущих: записывает в журнал данные о выполнении перехватываемых методов.

В следующем фрагменте кода содержится объявление новой аннотации. Здесь очень важно определить политику хранения с помощью аннотации `@Retention(RetentionPolicy.RUNTIME)`. По умолчанию в Java аннотации не могут перехватываться во время выполнения приложения. Нужно конкретно заявить, что нечто может перехватывать аннотации, определив политику хранения как `RUNTIME`. `@Target` определяет, для каких элементов языка может использоваться конкретная аннотация. По умолчанию ею можно сопровождать любые элементы,

но всегда лучше ограничить область применения только тем, для чего она создается, — в данном случае методами:

```
@Retention(RetentionPolicy.RUNTIME) ← Активировать аннотацию для перехвата
@Target(ElementType.METHOD) ← во время выполнения приложения
public @interface ToLog {
    Применять ее только
    } для методов
```

В листинге 6.10 представлен класс `CommentService`, в котором теперь определены три метода. Мы снабдим аннотацией только метод `deleteComment()`, и в результате, по идее, аспект будет перехватывать лишь данный метод.

Листинг 6.10. Определение трех методов в классе CommentService

```
@Service
public class CommentService {

    private Logger logger = Logger.getLogger(CommentService.class.getName());

    public void publishComment(Comment comment) {
        logger.info("Publishing comment:" + comment.getText());
    }

    @ToLog ← Мы создали аннотацию и отметили ею тот метод,
    public void deleteComment(Comment comment) { который должен перехватываться аспектом
        logger.info("Deleting comment:" + comment.getText());
    }

    public void editComment(Comment comment) {
        logger.info("Editing comment:" + comment.getText());
    }
}
```

Чтобы вплести аспект в методы, отмеченные новой аннотацией (рис. 6.13), мы воспользовались следующим выражением среза AspectJ: `@annotation(ToLog)`. Это выражение указывает на любой метод с аннотацией `@ToLog` (которую мы создали). В листинге 6.11 представлен класс аспекта, в котором использовано новое выражение среза, чтобы вплести логику аспекта в перехватываемые методы. Все очень просто, не так ли?

Листинг 6.11. Изменение выражения среза для вплетения аспекта в методы с новой аннотацией

```
@Aspect
public class LoggingAspect {

    private Logger logger = Logger.getLogger(LoggingAspect.class.getName());

    @Around("@annotation(ToLog)") ← Вплетение аспекта в методы с аннотацией @ToLog
    public Object log(ProceedingJoinPoint joinPoint) throws Throwable {
        // код метода
    }
}
```

```

@Service
public class CommentService {
    private Logger logger =
        Logger.getLogger(CommentService.class.getName());
    public void publishComment(Comment comment) {
        logger.info("Publishing comment:" + comment.getText());
    }
}

@Aspect
public class LoggingAspect {
    private Logger logger =
        Logger.getLogger(LoggingAspect.class.getName());
    @Around("@annotation(Tolog)")
    public Object log(ProceedingJoinPoint joinPoint) throws Throwable {
        // код метода
    }
}

@Tolog
public void deleteComment(Comment comment) {
    logger.info("Deleting comment:" + comment.getText());
}

public void editComment(Comment comment) {
    logger.info("Editing comment:" + comment.getText());
}
}

```

Вплетается в

Рис. 6.13. С помощью выражения среза `@Aspect` мы вплетаем логику аспекта в любой метод, отмеченный новой аннотацией. Это удобный способ отметить те методы, к которым должна применяться логика определенных аспектов

После запуска приложения будет перехватываться только метод с созданной нами аннотацией (в данном случае `deleteComment()`), и аспект будет выводить в консоль записи о выполнении этого метода. Там должно появиться примерно следующее:

```
Sep 29, 2020 2:22:42 PM services.CommentService publishComment
INFO: Publishing comment:Demo comment
Sep 29, 2020 2:22:42 PM aspects.LoggingAspect log
INFO: Method deleteComment with parameters [Comment{text='Demo comment',
➥ author='Natasha'}] will execute
Sep 29, 2020 2:22:42 PM services.CommentService deleteComment
INFO: Deleting comment:Demo comment
Sep 29, 2020 2:22:42 PM aspects.LoggingAspect log
INFO: Method executed and returned null
Sep 29, 2020 2:22:42 PM services.CommentService editComment
INFO: Editing comment:Demo comment
```

Аспект перехватывает только метод `deleteComment()`,
 который мы снабдили созданной нами аннотацией `@ToLog`

6.2.4. Другие полезные аннотации советов

Есть и другие аннотации советов, которые можно использовать с аспектами Spring. До сих пор я упоминал только аннотацию совета `@Around`. Она действительно встречается в приложениях Spring чаще других, поскольку покрывает практически все варианты реализации: с ее помощью можно что-то сделать до, после и даже вместо перехваченного метода. В аспекте можно изменить логику перехваченного метода любым способом.

Но на практике часто нужны далеко не все эти возможности сразу. Иногда имеет смысл поискать более быстрый способ реализовать желаемое. Любое приложение следует стремиться сделать как можно более простым. Избегая излишней сложности, вы получите продукт, более удобный в поддержке. Для простых сценариев в Spring предусмотрены еще четыре аннотации советов, не такие мощные, как `@Around`. Их рекомендуется использовать, когда возможностей этих аннотаций достаточно, чтобы упростить приложение.

Кроме `@Around`, в Spring есть следующие аннотации советов:

- `@Before` — вызывает метод, определяющий логику аспекта, перед выполнением перехватываемого метода;
- `@AfterReturning` — вызывает метод, определяющий логику аспекта, после успешного завершения перехватываемого метода. При этом значение,озвращаемое перехваченным методом, передается методу аспекта в качестве параметра. Если перехваченный метод выдаст исключение, метод аспекта не вызывается;
- `@AfterThrowing` — вызывает метод, определяющий логику аспекта, если перехваченный метод выдаст исключение. Экземпляр исключения передается методу аспекта в качестве параметра;

- `@After` — вызывает метод, определяющий логику аспекта, после выполнения перехватываемого метода, независимо от того, завершится ли перехваченный метод успешно или выдаст исключение.

Данные аннотации советов используются так же, как и `@Around`. Они сопровождаются выражением среза AspectJ, позволяющим вплести логику аспекта в выполнение соответствующих методов. При этом методы аспектов не получают параметр `ProceedingJoinPoint`, и поэтому с ними нельзя выбирать, в какой момент делегировать управление перехваченному методу. Теперь это событие происходит в соответствии с выбранной аннотацией (например, в случае `@Before` перехватываемый метод всегда выполняется после логики аспекта).

В проекте sq-ch6-ex5 вы найдете пример использования аспекта `@AfterReturning`. Образец применения аннотации `@AfterReturning` также показан в следующем фрагменте кода. Обратите внимание, что эта аннотация используется точно так же, как `@Around`.

```
@Aspect
public class LoggingAspect {
    private Logger logger = Logger.getLogger(LoggingAspect.class.getName());

    @AfterReturning(value = "@annotation(ToLog)", returning = "returnedValue")
    public void log(Object returnedValue) {
        logger.info("Method executed and returned " + returnedValue);
    }
}
```

Выражение среза AspectJ определяет, в какие методы будет вплетаться логика данного аспекта

Имя этого параметра должно совпадать со значением атрибута `returning`, переданного аннотации. Если же нас не интересует возвращаемое значение, данный параметр можно не указывать

При использовании `@AfterReturning` можно также получить значение, возвращаемое перехваченным методом. В этом случае к атрибуту `value`, соответствующему имени метода, добавляется атрибут `returning`

6.3. ЦЕПОЧКИ ВЫПОЛНЕНИЯ АСПЕКТОВ

До сих пор во всех рассмотренных примерах мы обращали внимание на то, что происходит, когда один аспект перехватывает выполнение метода. В реальных приложениях таких аспектов может быть несколько. Предположим, у нас есть метод, данные о выполнении которого нужно заносить в журнал и одновременно применять некие ограничения по безопасности. Для реализации этих действий обычно используются аспекты. Поэтому в данном случае у нас будут два аспекта, которые активируются при выполнении одного и того же метода. Мы можем создать сколько угодно аспектов, это совершенно нормально. Но тогда необходимо сформулировать для себя ответы на следующие вопросы.

- В какой последовательности Spring будет выполнять эти аспекты?
- Имеет ли значение порядок выполнения?

В примере ниже мы ответим на эти вопросы.

Итак, у нас есть метод, к которому необходимо применить некие ограничения по безопасности, а также заносить данные о выполнении этого метода в журнал. Для реализации этих обязанностей у нас есть следующие аспекты:

- **SecurityAspect** — применяет ограничения по безопасности. Данный аспект перехватывает метод, валидирует его вызов и, если не выполняются некие условия, не передает вызов дальше перехваченному методу (подробности работы **SecurityAspect** в данном случае не имеют значения; просто запомним, что он иногда не вызывает перехваченный метод);
- **LoggingAspect** — заносит в журнал сообщения о начале и завершении работы перехваченного метода.

Когда в один и тот же метод вплетаются несколько аспектов, они выполняются по очереди. Один из способов состоит в том, чтобы сначала подключался **SecurityAspect**, затем он передавал бы управление аспекту **LoggingAspect**, а тот, в свою очередь, делегировал контроль перехваченному методу. Второй вариант — сначала выполнить **LoggingAspect**, а затем делегировать управление **SecurityAspect**, который в итоге передаст контроль перехваченному методу. Так образуется цепочка выполнения аспектов.

Последовательность работы аспектов имеет значение, поскольку при их выполнении в другом порядке мы получим другие результаты. Так, в нашем примере **SecurityAspect** не всегда передает управление дальше; поэтому если вначале будет выполняться этот аспект, то до **LoggingAspect** дело будет доходить не всегда. Если мы ожидаем, что **LoggingAspect** будет регистрировать попытки выполнения метода, которые оказались неудачными из-за ограничений по безопасности, то такой вариант нам не подходит (рис. 6.14).

Ну хорошо, последовательность выполнения аспектов важна. Как тогда определить эту последовательность? По умолчанию Spring не гарантирует, что каждый раз при запуске приложения цепочка аспектов будет выполняться в одном и том же порядке. Если последовательность неважна, достаточно создать аспекты и позволить фреймворку задействовать их по своему усмотрению. Если же порядок выполнения аспектов имеет значение, то можно воспользоваться аннотацией **@Order**. Эта аннотация получает порядковый номер (число), соответствующий очередности выполнения данного аспекта в цепочке. Чем меньше это число, тем раньше заработает аспект. В случае двух одинаковых чисел очередность выполнения снова не будет определена. Рассмотрим применение аннотации **@Order** на примере.

В проекте sq-ch6-ex6 определены два аспекта, перехватывающие метод `publishComment()` бина `CommentService`. В листинге 6.12 представлен аспект **LoggingAspect**. Пока что порядок выполнения аспектов не определен.

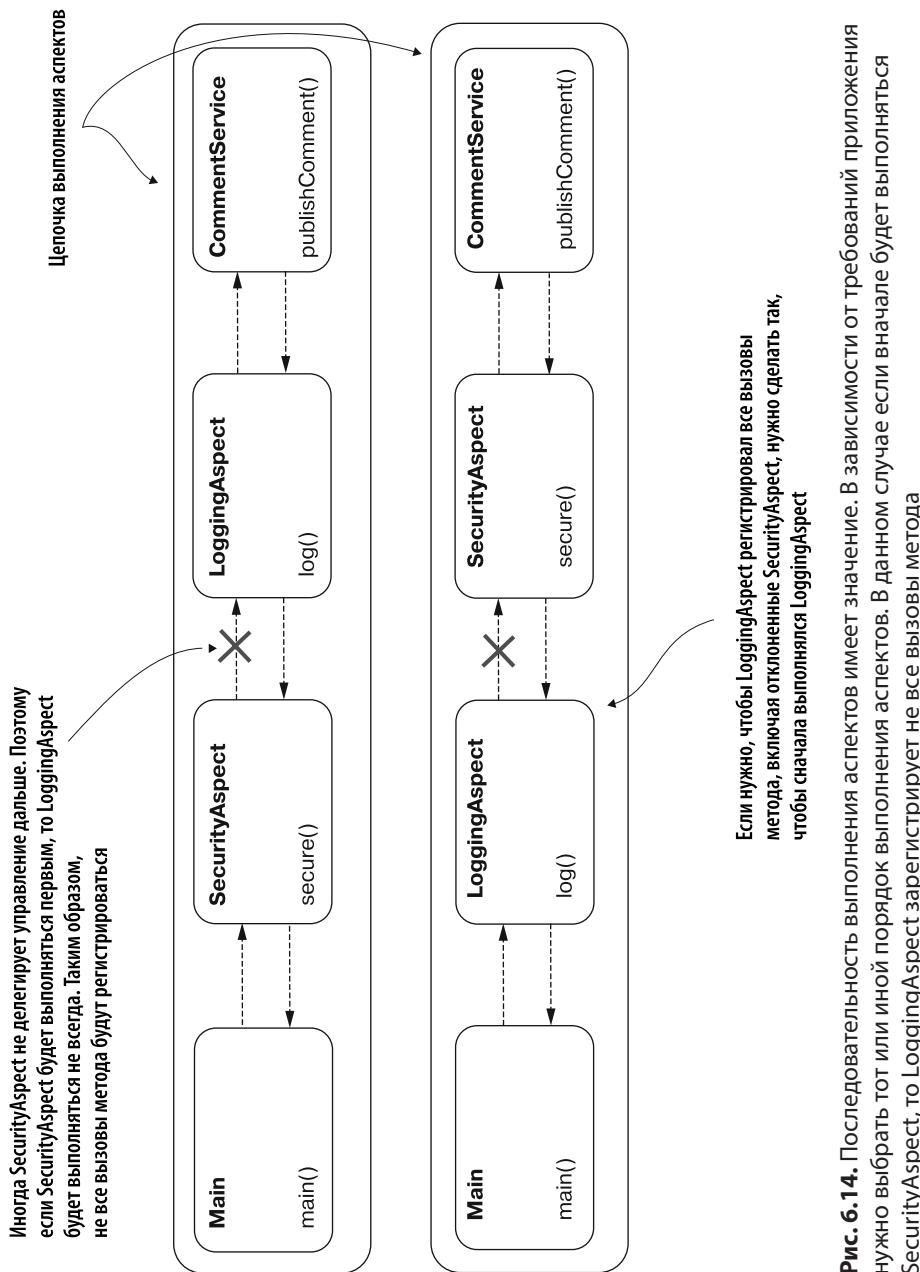


Рис. 6.14. Последовательность выполнения аспектов имеет значение. В зависимости от требований приложения нужно выбрать тот или иной порядок выполнения аспектов. В данном случае если вначале будет выполняться SecurityAspect, то LoggingAspect зарегистрирует не все вызовы метода

Листинг 6.12. Класс LoggingAspect

```

@Aspect
public class LoggingAspect {

    private Logger logger =
        Logger.getLogger(LoggingAspect.class.getName());

    @Around(value = "@annotation(ToLog)")
    public Object log(ProceedingJoinPoint joinPoint) throws Throwable {
        logger.info("Logging Aspect: Calling the intercepted method");

        Object returnedValue = joinPoint.proceed(); ←
            logger.info("Logging Aspect: Method executed and returned " +
                returnedValue);

        return returnedValue;
    }
}

```

Метод proceed() делегирует выполнение цепочки аспектов. Он вызывает либо следующий аспект, либо перехватываемый метод

Второй аспект в нашем примере называется **SecurityAspect**, и он представлен в листинге 6.13. Из соображений простоты, чтобы вы могли сконцентрироваться на главной теме нашего обсуждения, этот аспект не делает ничего особенного. Он, как и **LoggingAspect**, выводит сообщение в консоль — так мы можем легко убедиться, что метод выполняется.

Листинг 6.13. Класс SecurityAspect

```

@Aspect
public class SecurityAspect {

    private Logger logger =
        Logger.getLogger(SecurityAspect.class.getName());

    @Around(value = "@annotation(ToLog)")
    public Object secure(ProceedingJoinPoint joinPoint) throws Throwable {
        logger.info("Security Aspect: Calling the intercepted method");

        Object returnedValue = joinPoint.proceed(); ←
            logger.info("Security Aspect: Method executed and returned " +
                returnedValue);

        return returnedValue;
    }
}

```

Метод proceed() делегирует выполнение цепочки аспектов. Он вызывает либо следующий аспект, либо перехватываемый метод

Класс **CommentService** не отличается от используемого в предыдущих примерах. Но для удобства чтения он представлен в листинге 6.14.

Листинг 6.14. Класс CommentService

```

@Service
public class CommentService {

    private Logger logger =
        Logger.getLogger(CommentService.class.getName());

    @ToLog
    public String publishComment(Comment comment) {
        logger.info("Publishing comment:" + comment.getText());
        return "SUCCESS";
    }
}

```

Как мы помним, оба аспекта должны быть бинами, размещеными в контексте Spring. В данном примере я решил добавить аспекты в контекст с помощью аннотации `@Bean`. Мой класс конфигурации представлен в листинге 6.15.

Листинг 6.15. Объявление бинов аспектов в классе Configuration

```

@Configuration
@ComponentScan(basePackages = "services")
@EnableAspectJAutoProxy
public class ProjectConfig {

    @Bean ←
    public LoggingAspect loggingAspect() {
        return new LoggingAspect();
    }

    @Bean ←
    public SecurityAspect securityAspect() {
        return new SecurityAspect();
    }
}

```

Оба аспекта должны быть бинами, добавленными в контекст Spring

Метод `main()` вызывает метод `publishComment()` для бина `CommentService`. В моем случае результат выполнения приложения выглядит так:

```

Sep 29, 2020 6:04:22 PM aspects.LoggingAspect log
INFO: Logging Aspect: Calling the intercepted method
Sep 29, 2020 6:04:22 PM aspects.SecurityAspect secure
INFO: Security Aspect: Calling the intercepted method
Sep 29, 2020 6:04:22 PM services.CommentService publishComment
INFO: Publishing comment:Demo comment
Sep 29, 2020 6:04:22 PM aspects.SecurityAspect secure
INFO: Security Aspect: Method executed and returned SUCCESS
Sep 29, 2020 6:04:22 PM aspects.LoggingAspect log
INFO: Logging Aspect: Method executed and returned SUCCESS

```

Сначала вызывается аспект `LoggingAspect`, который передает управление аспекту `SecurityAspect`

Затем вызывается `SecurityAspect` и делегирует контроль перехваченному методу

Выполняется перехваченный метод

Перехваченный метод возвращает управление аспекту `SecurityAspect`

`SecurityAspect` возвращает управление `LoggingAspect`

Чтобы вам было проще понять последовательность появления записей в консоли, цепочка выполнения аспектов наглядно представлена на рис. 6.15.

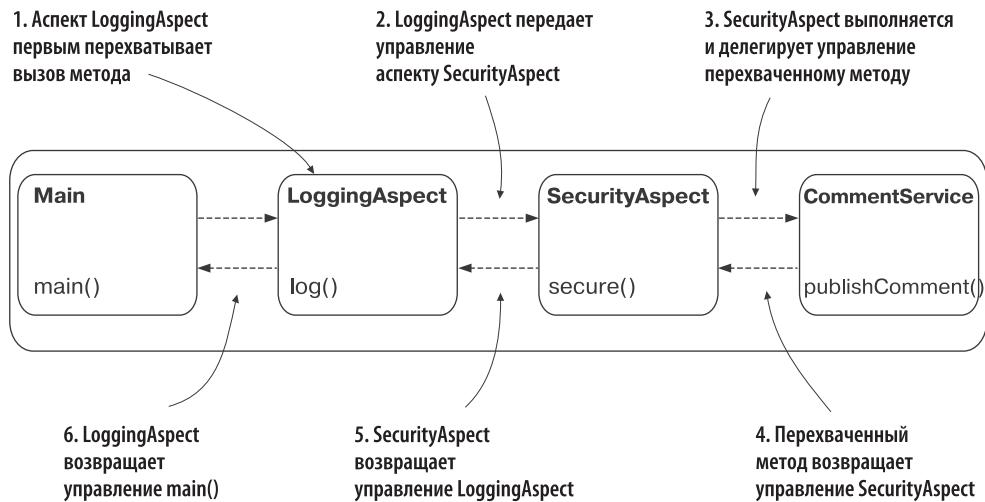


Рис. 6.15. Поток выполнения. LoggingAspect первым перехватывает вызов метода. Затем LoggingAspect передает управление по цепочке выполнения аспекту SecurityAspect, а тот, в свою очередь, делегирует вызов перехваченному методу. Перехваченный метод возвращает управление в SecurityAspect, а тот — в LoggingAspect

Чтобы аспекты LoggingAspect и SecurityAspect выполнялись в обратном порядке, воспользуемся аннотацией @Order. Посмотрите, как в следующем фрагменте кода с помощью @Order я задал позицию выполнения SecurityAspect (пример находится в проекте sq-ch6-ex7):

```

@Aspect
@Order(1) ←———— Определяет порядковый номер аспекта в цепочке выполнения
public class SecurityAspect {
    // код аспекта
}
  
```

Для LoggingAspect я также использовал @Order, чтобы присвоить этому аспекту более поздний номер в цепочке выполнения:

```

@Aspect
@Order(2) ←———— LoggingAspect будет выполняться вторым
public class LoggingAspect {
    // код аспекта
}
  
```

Запустив приложение еще раз, увидим, что последовательность выполнения аспектов изменилась. Теперь записи в консоли должны выглядеть так:

```
Sep 29, 2020 6:38:20 PM aspects.SecurityAspect secure
INFO: Security Aspect: Calling the intercepted method
Sep 29, 2020 6:38:20 PM aspects.LoggingAspect log
INFO: Logging Aspect: Calling the intercepted method
Sep 29, 2020 6:38:20 PM services.CommentService publishComment
INFO: Publishing comment:Demo comment
Sep 29, 2020 6:38:20 PM aspects.LoggingAspect log
INFO: Logging Aspect: Method executed and returned SUCCESS
Sep 29, 2020 6:38:20 PM aspects.SecurityAspect secure
INFO: Security Aspect: Method executed and returned SUCCESS
```

SecurityAspect первым перехватывает вызов метода и передает управление LoggingAspect

LoggingAspect выполняется и делегирует контроль перехваченному методу

Перехваченный метод выполняется и возвращает управление LoggingAspect

LoggingAspect возвращает управление SecurityAspect

SecurityAspect возвращає управление методу main(), который изначально вызвал перехватываемый метод

Чтобы лучше понять, почему записи в консоли появляются именно в такой последовательности, цепочка выполнения наглядно представлена на рис. 6.16.

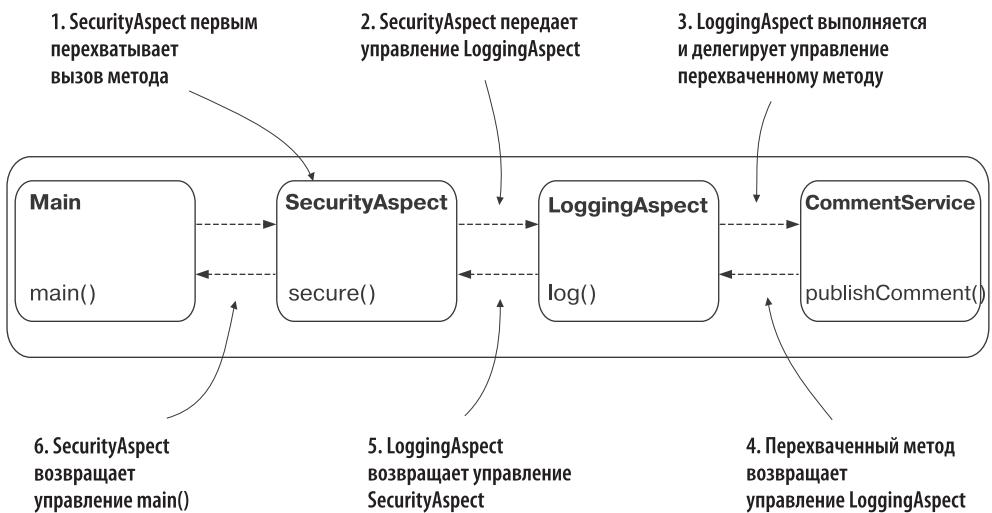


Рис. 6.16. Поток выполнения приложения после изменения последовательности применения аспектов: SecurityAspect первым перехватывает вызов метода и передает управление аспекту LoggingAspect, который, в свою очередь, делегирует контроль перехваченному методу. Перехваченный метод возвращает управление аспекту LoggingAspect, а тот — аспекту SecurityAspect

РЕЗЮМЕ

- Аспект — это объект, перехватывающий вызов метода и выполняющий некоторую логику до, после или даже вместо выполнения перехваченного метода. Это позволяет отделить часть кода от бизнес-логики, благодаря чему приложение становится проще в обслуживании.
- Используя аспект, можно написать логику, которая будет выполняться вместе с методом, но размещаться отдельно от него. Таким образом, когда кто-нибудь будет читать код, он увидит только то, что имеет отношение к бизнес-логике приложения.
- Аспекты — опасный инструмент. Код, перегруженный ими, лишь усложняет обслуживание приложения. Не используйте аспекты где попало. Принимая решение об их применении, убедитесь, что они действительно улучшат реализацию.
- Аспекты лежат в основе многих важных функций Spring, таких как транзакции и методы обеспечения безопасности.
- Чтобы создать аспект в Spring, нужно добавить к классу, в котором описана логика аспекта, аннотацию `@Aspect`. Однако, поскольку Spring должен управлять экземпляром этого класса, нужно также не забыть создать бин этого типа и поместить его в контекст Spring.
- Чтобы сообщить Spring, какие методы должен перехватывать аспект, используются выражения срезов `AspectJ`. Эти конструкции служат значениями для аннотаций советов. В Spring есть пять аннотаций советов: `@Around`, `@Before`, `@After`, `@AfterThrowing` и `@AfterReturning`. Чаще всего применяется `@Around`, как наиболее универсальная.
- Вызов одного и того же метода может перехватываться несколькими аспектами. В данном случае рекомендуется задавать последовательность выполнения аспектов с помощью аннотации `@Order`.

Часть II

Реализация

Во второй части вы научитесь на практике применять возможности Spring, которые будут часто нужны в реальных приложениях. Вначале мы рассмотрим веб-приложения, а затем вы будете передавать данные между приложениями и работать с хранилищами данных. Вы узнаете, что благодаря Spring это делается просто и легко. В завершение книги мы напишем модуль и интеграционные тесты для функционала созданных вами Spring-приложений.

Навыки, которые вы получите в этой части, требуют понимания основных концепций Spring — контекста и аспектов. Если вы уже знаете, как работают контекст и аспекты Spring, и горите желанием поскорее начать писать приложения с использованием фреймворка, то можете начать чтение книги сразу со второй части. Однако если с основами вы себя чувствуете еще не очень уверенно, то лучше вернитесь в первую часть и изучите эти темы.



Введение в Spring Boot и Spring MVC

В этой главе

- ✓ Как создать веб-приложение.
- ✓ Как использовать Spring Boot при разработке Spring-приложений.
- ✓ Что такое архитектура Spring MVC.

После того как вы освоили все необходимые основы Spring, можно обратить внимание на веб-приложения и на то, как использовать Spring при их разработке. Функции Spring, которые мы обсуждали ранее, можно применять в любом приложении. Но большинство продуктов, разрабатываемых на основе Spring, — это веб-приложения. В главах 1–6 мы рассмотрели контекст и аспекты Spring — темы, совершенно необходимые для понимания дальнейшей информации в книге (включая содержимое данной главы). Если вы перешли сразу к этой главе, но еще не знаете, как работать с контекстом и аспектами Spring, последующее обсуждение может оказаться для вас слишком сложным. Прежде чем вы продолжите читать книгу, настоятельно рекомендую убедиться, что вы хорошо понимаете основы работы с фреймворком.

Spring сильно упрощает разработку веб-приложений. Начнем главу с выяснения, что такое веб-приложения и как они работают.

Для создания веб-приложений мы воспользуемся проектом экосистемы Spring, который называется Spring Boot. Мы рассмотрим его в разделе 7.2 — и вы узнаете, почему этот проект играет такую важную роль в создании приложений. В разделе 7.3 мы обсудим стандартную архитектуру простого веб-приложения

на основе Spring и построим его с использованием Spring Boot. Дочитав эту главу, вы поймете, как работают веб-приложения, и будете способны создать простейшее веб-приложение на основе Spring.

Главная цель главы — помочь вам понять базу, на которой строятся веб-приложения. В главах 8 и 9 вы научитесь использовать главные возможности Spring, применяемые в большинстве реальных веб-приложений. Но все описанное в последующих главах основано на принципах, раскрываемых здесь.

7.1. ЧТО ТАКОЕ ВЕБ-ПРИЛОЖЕНИЕ

Что собой представляет веб-приложение? Вы совершенно точно используете такие продукты каждый день. Возможно, прямо сейчас, начиная читать эту главу, вы оставили в браузере несколько открытых вкладок. Может быть, вы вообще читаете данную книгу не в бумажной версии, а в приложении Manning liveBook.

Любое приложение, которое вы открываете в веб-браузере, — это веб-приложение. Было время, когда десктопные программы, установленные на компьютере, использовались почти для всего (рис. 7.1). Но прошли годы, и большинство приложений теперь доступны через браузер. Благодаря этому они стали гораздо удобнее. Больше ничего не нужно устанавливать: сегодня эти приложения доступны с любого устройства, подключенного к интернету, например с планшета или смартфона.

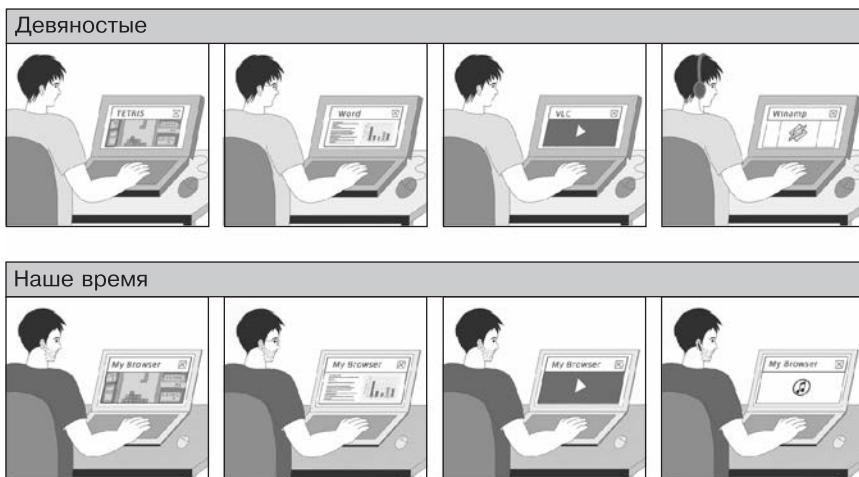


Рис. 7.1. Времена изменились. В 1990-х мы постоянно пользовались десктопными программами. Сейчас практически все нужные нам приложения — это веб-приложения. Вам, как разработчику, крайне важно уметь их создавать

Я хочу, чтобы у вас после прочтения данного раздела сформировалась ясная картина того, что вы будете создавать. Что такое веб-приложение, что нужно

сделать, чтобы создать такой продукт и запустить его? Как только у вас будет четкое представление об этом, мы разработаем такое приложение с использованием Spring.

7.1.1. Основные сведения о веб-приложениях

Рассмотрим в общих чертах, что такое веб-приложение с технической точки зрения. Данный обзор позволит нам впоследствии увидеть шире наши возможности при создании веб-приложений.

Итак, любое веб-приложение состоит из двух частей, таких как.

- **клиентская часть** — то, с чем непосредственно взаимодействует пользователь. Представлена веб-браузером. Браузер отправляет запросы на веб-сервер, получает от него ответы и предоставляет пользователю способ взаимодействия с приложением. Клиентскую часть приложения еще называют *фронтиеном*;
- **серверная часть** — получает запросы от клиента и отправляет ему в ответ данные. Серверная часть содержит логику, которая обрабатывает и иногда сохраняет запрашиваемые клиентом данные перед тем, как отправить ему ответ. Серверную часть также называют *бэкеном*.

Общая схема веб-приложения представлена на рис. 7.2.

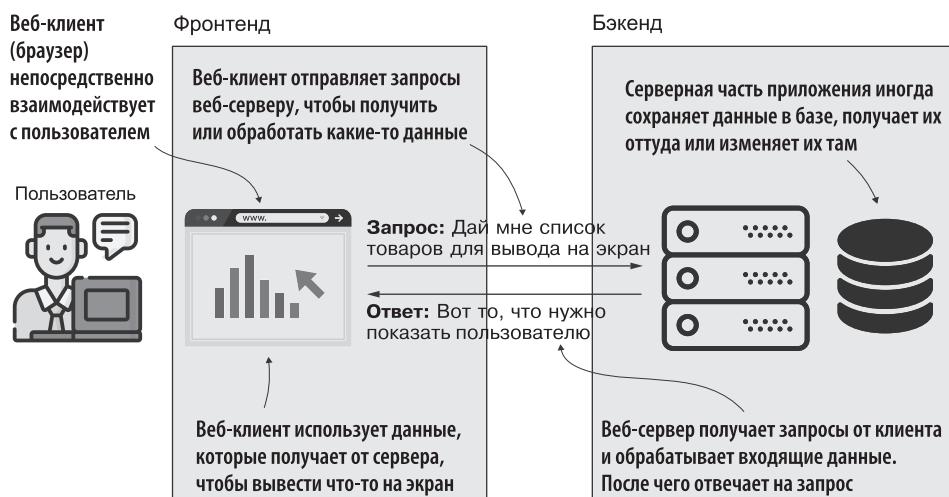


Рис. 7.2. Общая схема работы веб-приложения. Пользователь взаимодействует с приложением через фронтенд. Фронтенд обменивается информацией с бэкендом, чтобы выполнить логику по запросу пользователя и получить данные для вывода на экран. Бэкенд выполняет бизнес-логику, иногда сохраняет данные в базе или обменивается информацией с внешними сервисами

Говоря о веб-приложениях, мы обычно имеем в виду клиентскую и серверную части. Но важно учитывать, что бэкенд обслуживает сразу нескольких клиентов на конкурентной основе. Множество людей могут использовать одно и то же веб-приложение в одно и то же время на разных платформах. Пользователи могут открывать приложение в браузере на компьютере, в телефоне, на планшете и т. п. (рис. 7.3).

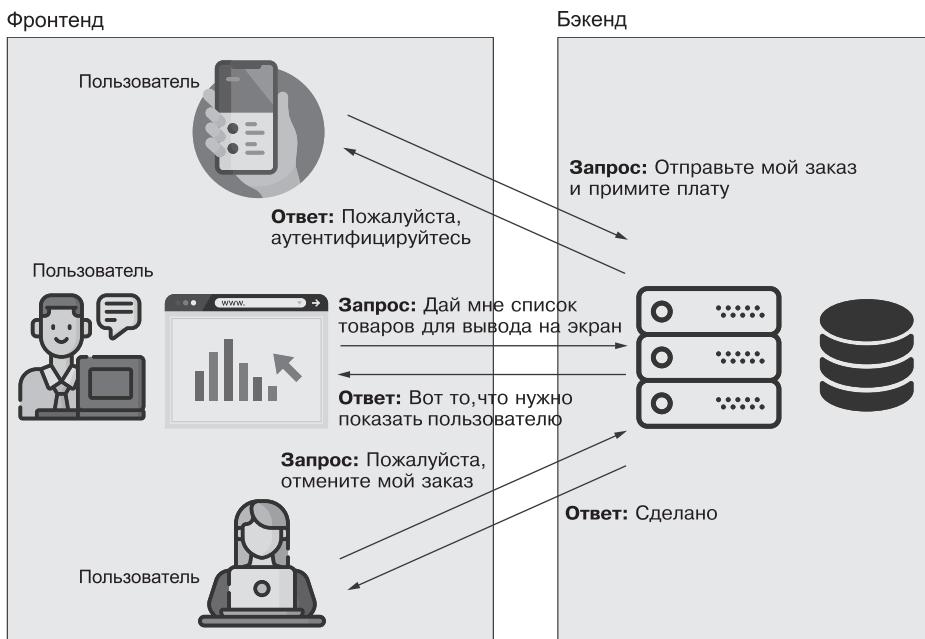


Рис. 7.3. Говоря о веб-приложениях, мы обычно упоминаем клиента в единственном числе. Однако необходимо учитывать, что к одному и тому же приложению обращается сразу много пользователей через веб-браузеры. Каждый из них делает свои запросы, которые требуют соответствующих действий. Следовательно, некоторые операции на стороне бэкенда выполняются конкурентно — это важно понимать. Если вы напишете код, позволяющий сразу нескольким объектам обращаться к одному ресурсу и изменять его, то приложение может работать неправильно из-за состояния гонки

7.1.2. Способы реализации веб-приложений на основе Spring

Далее мы обсудим два основных способа реализации веб-приложений. В главах 8–10 мы применим их все для создания приложения и рассмотрим особенности каждого из них. Однако сейчас мне бы хотелось, чтобы вы просто знали о возможностях выбора и имели общие представления о вариантах. Важно понимать, как создаются веб-приложения, чтобы потом не запутаться при выполнении примеров.

Веб-приложения по способу функционирования бывают разные.

- Приложения, в которых бэкенд в ответ на запрос клиента дает полностью готовое представление.* В таких продуктах браузер получает данные от бэкенда и сразу выводит их на экран пользователя. Ниже мы рассмотрим этот способ и проверим, как он работает, создав простое приложение. Затем, в главах 8 и 9, мы обсудим более тонкие нюансы данного подхода, возникающие в реальных приложениях.
- Приложения с разделением обязанностей между фронтендом и бэкендом.* В таких продуктах бэкенд предоставляет только первичные данные. Получив ответ от бэкенда, браузер не выводит эти данные сразу на экран, а запускает некое особые клиентское приложение, которое получает ответы от сервера, обрабатывает данные и сообщает браузеру, что именно нужно вывести. Мы рассмотрим этот способ создания приложений и выполним соответствующие примеры в главе 9.

Первый вариант, где в приложении нет четкого разделения на клиентскую и серверную части, показан на рис. 7.4. В таких приложениях практически все происходит на стороне бэкенда. Бэкенд получает запросы, отражающие действия пользователя, и выполняет некую логику. После этого сервер возвращает браузеру нужную информацию для вывода на экран. Бэкенд передает данные в тех форматах, которые браузер может интерпретировать и вывести, — HTML, CSS, изображения и т. п. Сервер также может предоставить сценарии, написанные на языках программирования, которые браузер способен интерпретировать и выполнить (таких как JavaScript).

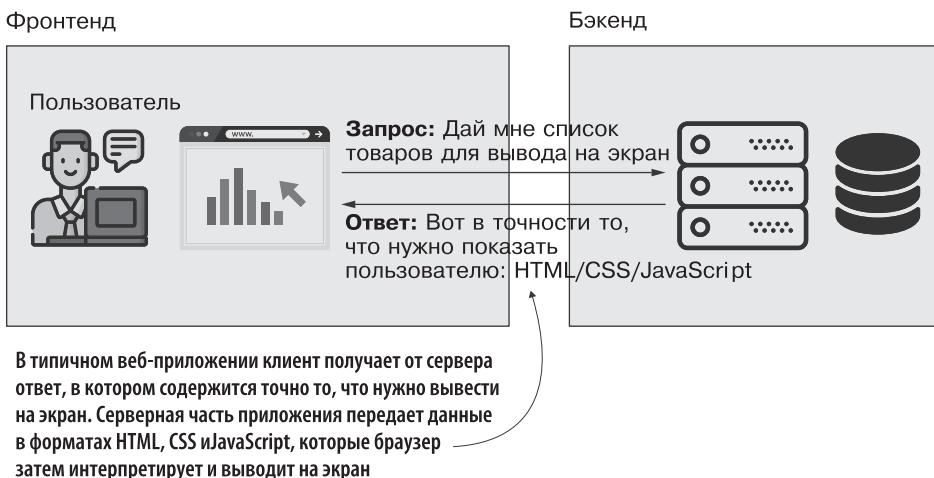


Рис. 7.4. Если в приложении нет разделения на серверную и клиентскую части, браузер просто выводит на экран то, что ему дает бэкенд. Сервер получает запросы от браузера, выполняет какую-то логику и возвращает ответ. В качестве ответа бэкенд предоставляет содержимое в виде HTML, CSS, а также в других форматах, которые браузер способен интерпретировать и вывести на экран

На рис. 7.5 показано приложение, в котором используется разделение на клиентскую и серверную части. Сравните ответ сервера на рис. 7.5 с ответом на рис. 7.4: теперь бэкенд вместо того, что нужно в точности вывести на экран, передает браузеру только первичные данные. Браузер запускает независимое клиентское приложение, которое загружается с сервера в ответ на первый запрос. Приложение получает первичные данные от бэкенда, интерпретирует их и принимает решение о том, как именно следует вывести информацию. В главе 9 мы рассмотрим данный способ функционирования веб-приложений более подробно.

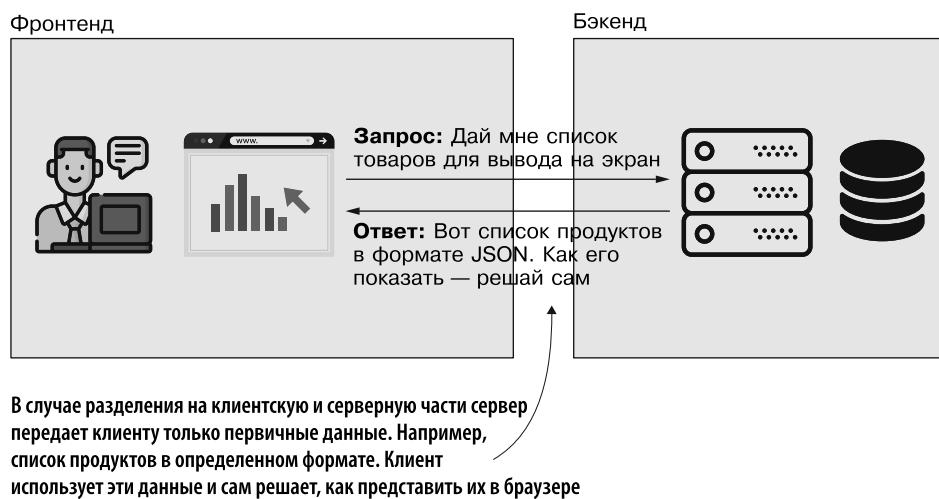


Рис. 7.5. Разделение на клиентскую и серверную части. В ответе сервера не содержатся точно те данные, которые нужно вывести в браузере. Бэкенд передает информацию, но не сообщает, как браузер должен ее представить и вообще что нужно с ней делать. Теперь сервер отправляет только первичные данные (обычно представленные в легко анализируемом формате, таком как JSON или XML). Браузер запускает клиентское приложение, которое получает первичный ответ от сервера, обрабатывает его и выводит данные

На практике встречаются оба варианта. Иногда разработчики называют подход с делением приложений на клиентскую и серверную части современным. Такой вариант упрощает разработку крупных приложений, поскольку реализацией разных частей занимаются разные команды. Это позволяет задействовать больше специалистов. Кроме того, клиентская и серверная части также могут развертываться независимо друг от друга, что для крупных приложений является значительным преимуществом.

Другой подход, при котором деления на клиентскую и серверную части нет, применяется в основном для небольших приложений. Мы подробно рассмотрим оба варианта, а потом я покажу преимущества каждого, чтобы вы сами в зависимости от своих потребностей могли решать, когда и какой из них следует выбрать.

7.1.3. Использование контейнера сервлетов в веб-разработке

Проанализируем более подробно, что и зачем нужно для создания веб-приложения на основе Spring. Вы уже знаете, что у веб-приложения есть фронтенд и бэкенд. Однако еще не умеете реализовывать веб-приложения на базе Spring. Но, разумеется, наша цель заключается как раз в этом — выработав главные навыки работы со Spring, научиться создавать приложения на его основе, так что пора двигаться дальше и выяснить, что нам для этого понадобится.

Один из важнейших моментов, который следует учитывать, — это коммуникация между клиентом и сервером. Веб-браузер обменивается данными с сервером по сети через протокол передачи гипертекста — Hypertext Transfer Protocol (HTTP), — который аккуратно регламентирует данный процесс. Но вам, если только вы не увлекаетесь информационными сетями, для создания веб-приложений не нужно знать все нюансы HTTP. Как разработчику программного обеспечения, вам достаточно факта, что этот протокол используется компонентами веб-приложения для обмена данными по принципу «запрос — ответ», когда клиент ожидает ответа на каждый отправленный запрос. В приложении В вы найдете всю необходимую информацию об HTTP для понимания того, о чем пойдет речь в главах 7–9.

Но значит ли это, что приложение должно уметь обрабатывать HTTP-сообщения? В принципе, при желании можно реализовать такую функцию. Но лучше воспользоваться уже существующим компонентом для поддержки HTTP (если, конечно, вы не хотите немного развлечься написанием низкоуровневого функционала).

В сущности, вам нужно не только что-то понимающее HTTP-запросы, но и что-то способное передать HTTP-запрос и получить ответ для Java-приложения. Это «что-то» называется *контейнером сервлетов* (для которого еще иногда употребляют термин «веб-сервер»). Контейнер сервлетов служит для Java-приложения переводчиком с HTTP. Таким образом, нет необходимости реализовывать коммуникационный уровень в продукте. Одним из самых популярных контейнеров сервлетов считается Tomcat, который также является одной из зависимостей, использованных в примерах этой книги.

ПРИМЕЧАНИЕ

В издании я использую Tomcat, но вы при разработке Spring-приложений можете воспользоваться любой из его альтернатив. Решений, применяемых в реальных приложениях, очень много. В их число входят, например, Jetty (<https://www.eclipse.org/jetty/>), Jboss (<https://www.jboss.org/>) и Payara (<https://www.payara.fish/>).

На рис. 7.6 показано, какое место занимает контейнер сервлетов (Tomcat) в архитектуре приложения.

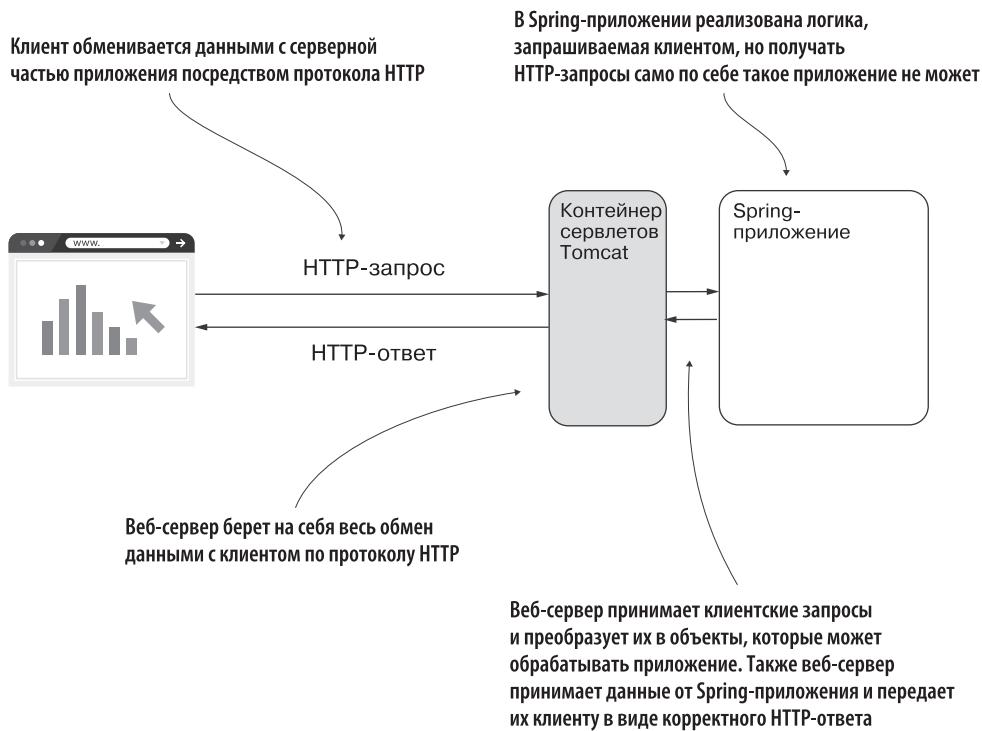


Рис. 7.6. Контейнер сервлетов (такой как Tomcat) умеет «говорить на языке HTTP». Он переводит HTTP-запрос для Spring-приложения, а ответ приложения — в HTTP-ответ. Благодаря этому можно не беспокоиться, какой протокол используется для передачи данных по сети, — достаточно представить все в виде объектов и методов Java

Но если функции контейнера сервлетов этим и ограничиваются, почему он называется контейнером *сервлетов*? Что такое сервлеты?

Сервлет — это просто объект Java, который напрямую взаимодействует с контейнером сервлетов. Когда контейнер получает HTTP-запрос, он вызывает метод объекта сервлета, которому передает этот запрос в виде параметра. Сервлет отправляет ответ клиенту, сделавшему запрос, а метод получает параметр, представляющий собой HTTP-ответ.

Было время, когда, с точки зрения разработчика, сервлеты являлись главными компонентами бэкенда веб-приложений. Допустим, специалисту требовалось

написать для веб-приложения новую страницу, расположение которой указывалось определенным URL (например, /home/profile/edit). Для этого ему приходилось создавать новый экземпляр сервлета, описывать его конфигурацию в контейнере и назначать заданный путь (рис. 7.7). В сервлете содержалась логика, позволяющая принимать пользовательский запрос и готовить ответ, включая информацию для браузера о том, как выводить данный ответ на экран. Для любого пути, по которому мог обратиться клиент, разработчик должен был создать в контейнере сервлетов отдельный экземпляр со своей конфигурацией. Этот компонент управляет экземплярами сервлетов, добавленными в контекст, поэтому мы и назвали его контейнером сервлетов. В сущности, у него есть контекст экземпляров, которые он контролирует, подобно тому как Spring управляет бинами в своем контексте. Поэтому компоненты наподобие Tomcat мы называем контейнерами сервлетов.

Как вы узнаете далее, экземпляры сервлетов обычно не приходится разрабатывать. Мы будем использовать сервлеты в приложениях, создаваемых на основе Spring, но писать сами сервлеты нам не придется, поэтому нет смысла тратить внимание на изучение этого момента. Главное — запомнить, что сервлет — это точка входа в логику приложения, компонент, с которым напрямую взаимодействует контейнер сервлетов (в данном случае Tomcat). С его помощью данные из клиентского запроса попадают в приложение, а ответ возвращается клиенту (рис. 7.8).

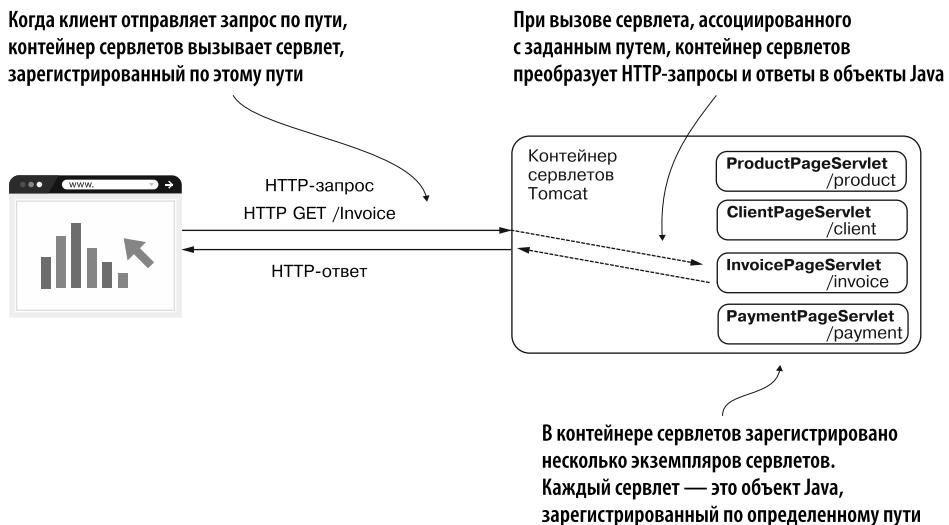


Рис. 7.7. Контейнер сервлетов (Tomcat) регистрирует несколько экземпляров сервлетов. У каждого из них есть свой путь. Когда клиент отправляет запрос, Tomcat вызывает метод сервлета, связанного с путем, по которому клиент сделал запрос. Сервлет получает значения запроса и формирует ответ, который затем Tomcat возвращает клиенту

В веб-приложении Spring создается объект сервлета. Мы регистрируем этот объект, так что Tomcat может вызвать его для любого пути, указанного в клиентском запросе. Сервлет играет роль точки входа в логику приложения

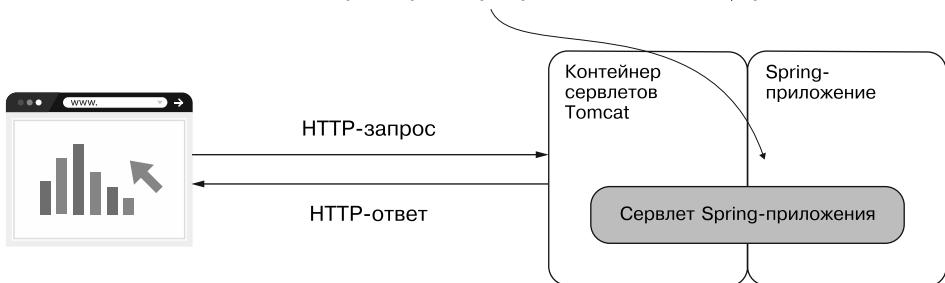


Рис. 7.8. После того как объект сервлета определен в Spring-приложении и зарегистрирован в контейнере сервлетов, и Spring, и контейнер знают о его существовании и могут им управлять. Контейнер вызывает сервлет для любого клиентского запроса, чтобы сервлет управлял этим запросом и формировал ответ

7.2. МАГИЯ SPRING BOOT

Чтобы создать веб-приложение Spring, нужно описать конфигурацию контейнера сервлетов, создать экземпляр сервлета и убедиться, что его настройки позволяют Tomcat вызывать этот экземпляр для любого запроса. До чего же хлопотно писать все это! Много лет назад, когда я преподавал Spring 3 (последняя версия фреймворка на тот момент) и мы писали конфигурации для таких приложений, данную часть курса в равной степени ненавидели и студенты, и я. К счастью, времена изменились, и мне больше не придется этим вас напрягать.

Ниже мы рассмотрим Spring Boot — инструмент для создания современных Spring-приложений. В настоящее время Spring Boot является одним из самых важных проектов экосистемы Spring. Он ускоряет создание Spring-приложений, позволяя сконцентрироваться на бизнес-логике и избавляя от написания огромных кусков кода, ответственного за конфигурацию. Эта возможность особенно полезна для сервисно-ориентированных архитектур (СОА) и микросервисов, для которых создается большинство приложений (об этом читайте в приложении А).

Ниже я приведу список самых важных, по моему мнению, опций Spring Boot и тех возможностей, которые они открывают.

- *Упрощенное создание проектов.* Сервис инициализации проекта создает пустую, но полностью сконфигурированную заготовку приложения.
- *Диспетчеры зависимостей.* Spring Boot группирует зависимости, используемые для тех или иных целей, и объединяет их с помощью диспетчеров. Больше не нужно выяснять ни полный список зависимостей, которые необходимо добавить в проект с определенной целью, ни то, какие их версии нужны для обеспечения совместимости.

- *Автоконфигурация на основе зависимостей.* На основе зависимостей, добавленных в проект, Spring Boot создает несколько конфигураций по умолчанию. Вместо того чтобы задавать все эти конфигурации самому, теперь достаточно изменить отдельные из них, которые не соответствуют вашим потребностям, причем для этого приходится писать меньше кода (или не приходится его писать вообще).

Обсудим перечисленные возможности Spring Boot более подробно и научимся их применять. И в качестве примера напишем наше первое веб-приложение на базе Spring.

7.2.1. Создание проекта Spring Boot с помощью сервиса инициализации проекта

Далее мы рассмотрим использование сервиса инициализации проекта для создания проекта Spring Boot. Некоторые считают, что данный сервис не заслуживает особого внимания, однако лично я не могу выразить словами, как счастлив, что он существует. Вам как разработчику не придется делать по несколько проектов в день, так что вы не сможете оценить главное преимущество этой функции. Но для студентов и преподавателей, которым приходится ежедневно писать много проектов Spring Boot, эта функция экономит огромное количество часов однообразной, бессмысленной работы, которую приходилось бы выполнять, если бы проект создавался с нуля. Чтобы понять, как сервис инициализации может вам помочь, воспользуемся им и создадим проект sq-ch7-ex1.

В некоторых IDE сервис инициализации проектов интегрирован непосредственно, в других — нет. Например, в IntelliJ Ultimate или STS он доступен при создании проекта (рис. 7.9), а в IntelliJ Community такая возможность отсутствует.

Если ваша IDE поддерживает этот сервис, то вы его найдете, скорее всего, в меню создания проекта в виде команды **Spring Initializr**. Если же он не интегрирован в IDE напрямую, можно воспользоваться сервисом непосредственно через браузер, по адресу <http://start.spring.io>. Сервис инициализации помогает создать проект, который затем можно будет импортировать в любую IDE. Воспользуемся этой возможностью для разработки нашего первого проекта. Ниже перечислены операции, которые нужно выполнить, чтобы создать проект Spring Boot с помощью start.spring.io (рис. 7.10).

1. Откройте в браузере страницу start.spring.io.
2. Укажите свойства проекта (язык, версию, систему сборки и т. п.).
3. Укажите зависимости, которые нужно добавить в проект.
4. Загрузите упакованный проект с помощью кнопки **GENERATE**.
5. Распакуйте проект и откройте его в IDE.

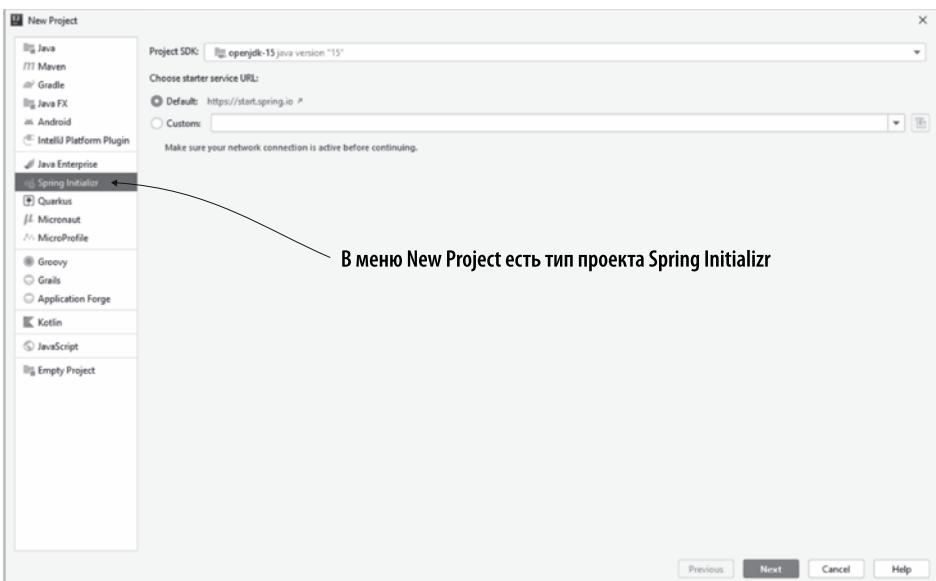


Рис. 7.9. Сервис инициализации проектов интегрирован в некоторые IDE. Например, в IntelliJ Ultimate, чтобы создать приложение Spring Boot с использованием данной возможности, достаточно выбрать из меню New Project команду Spring Initializr



Рис. 7.10. Операции, необходимые для того, чтобы сгенерировать проект Spring Boot с помощью start.spring.io: войдите на страницу start.spring.io, укажите свойства проекта и необходимые зависимости и загрузите упакованный проект. Затем откройте его в браузере

Открыв в браузере страницу start.spring.io, вы увидите интерфейс, подобный показанному на рис. 7.11. Здесь нужно указать некоторые свойства проекта, такие как инструмент сборки (Maven или Gradle) и используемую версию Java. Spring Boot даже позволяет изменить синтаксис приложения на Kotlin или Groovy.

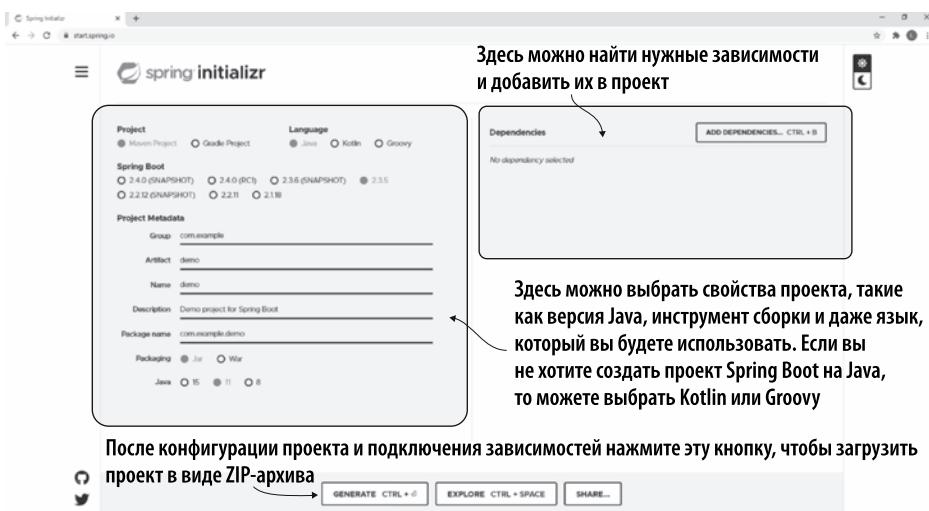


Рис. 7.11. Интерфейс start.spring.io. Открыв страницу start.spring.io, вы сможете выбрать основные параметры конфигурации, установить зависимости и загрузить упакованный проект

Spring Boot предлагает много вариантов, но мы для согласованности примеров будем по-прежнему использовать Maven и Java 11. На рис. 7.12 показано заполнение полей для нашего нового проекта Spring Boot. В этом примере нам понадобится только зависимость, которая называется Spring Web. Она добавляет в проект все, что нужно для веб-приложения на основе Spring.

После того как вы щелкнете на кнопке GENERATE, браузер загрузит ZIP-архив с проектом Spring Boot. Теперь посмотрим, какие главные свойства Spring Initializr вписывает в конфигурацию проекта Maven (рис. 7.13):

- класс `main` Spring-приложения;
- блок `<parent>` для Spring Boot POM;
- зависимости;
- плагин Spring Boot Maven;
- файл свойств.

Полезно иметь представление о том, как выглядит ваш проект. Поэтому рассмотрим подробнее каждый элемент конфигурации.

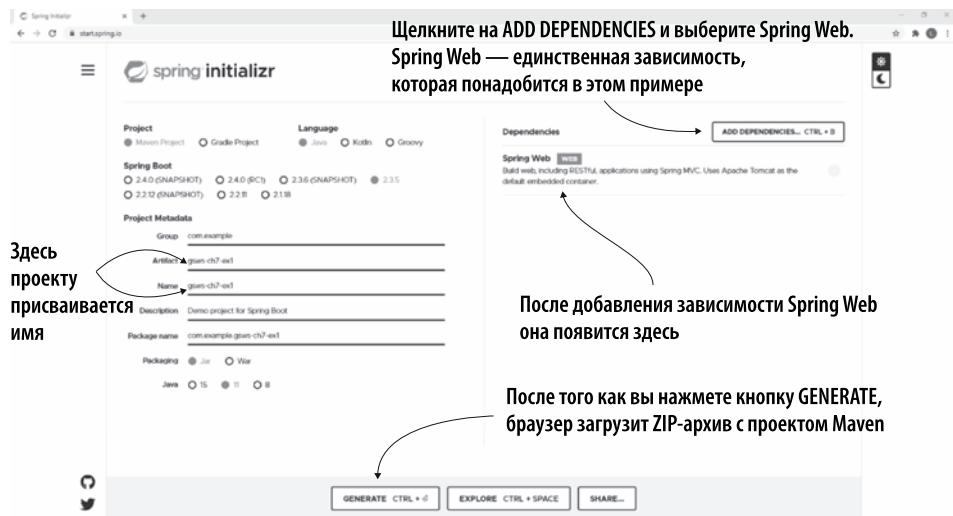


Рис. 7.12. В нашем примере нужно добавить зависимость Spring Web. Для этого воспользуйтесь кнопкой Add Dependencies, которая находится в правом верхнем углу окна. Необходимо также дать проекту имя

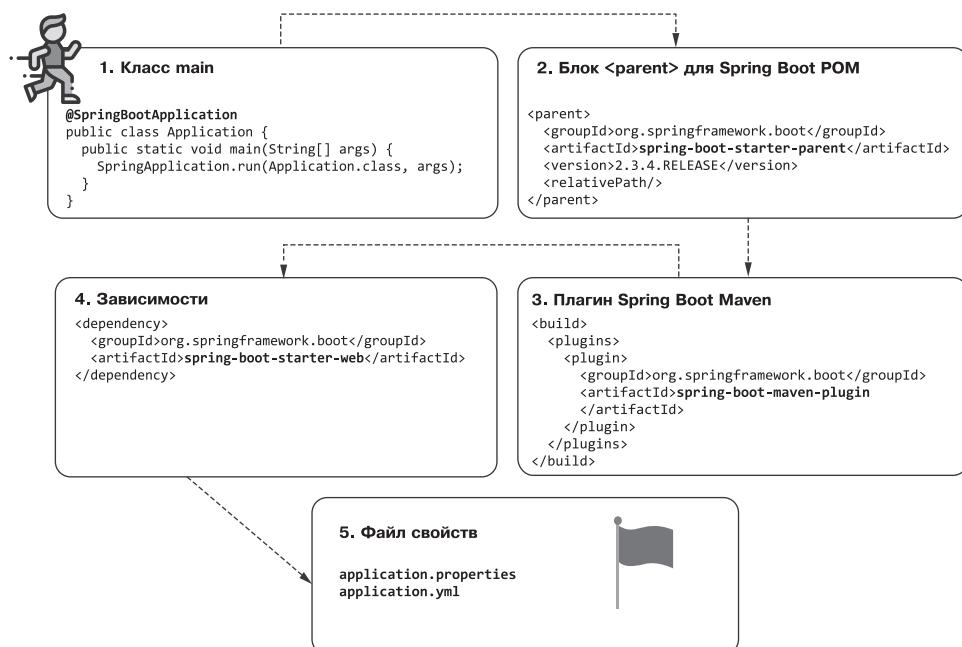


Рис. 7.13. Когда Spring Initializr генерирует проект Spring Boot, он настраивает некоторые конфигурационные параметры, чего обычно не происходит при создании проекта в Maven

Класс main, созданный в start.spring.io

Прежде всего рассмотрим главный класс приложения. Распакуйте загруженный файл и откройте его в IDE. Вы увидите, что Spring Initializr создал класс `Main` и добавил некоторые параметры в файл конфигурации `pom.xml`. Класс `Main` приложения Spring Boot снабжен аннотацией `@SpringBootApplication` и выглядит примерно так:

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

Аннотация объявляет, что данный класс является главным классом приложения Spring Boot

Этот код был сгенерирован Spring Initializr. В книге мы будем обращать внимание только на то, что имеет отношение к рассматриваемым примерам. В частности, я не буду подробно объяснять, что делает метод `SpringApplication.run()` и как именно аннотация `@SpringBootApplication` используется в Spring Boot. Все эти нюансы нам сейчас неважны. Spring Boot – тема отдельного издания. Но в какой-то момент вы, без сомнения, захотите подробно разобраться, как именно он работает. На этот случай рекомендую вам следующие книги: Craig Walls, *Spring Boot in Action* (Manning, 2015) и «Spring Boot по-быстрому» Марка Хеклера (Питер, 2022).

Блок <parent> для Spring Boot Maven, созданный в start.spring.io

Теперь обратим внимание на файл `pom.xml` созданного проекта. Открыв его, вы заметите, что сервис инициализации добавил туда несколько элементов. Один из самых важных – родительский узел Spring Boot, который выглядит примерно так:

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.3.4.RELEASE</version>
    <relativePath/>
</parent>
```

Одна из основных вещей, которые делает родительский узел, – это сообщение о совместимых версиях зависимостей, которые будут добавляться в проект. Мы, как правило, не будем указывать версии зависимостей сами. Вместо этого мы (в соответствии с рекомендациями) позволим Spring Boot автоматически выбирать нужные, чтобы не возникло несовместимостей.

Плагин Spring Boot Maven, настраиваемый start.spring.io

Теперь рассмотрим плагин Spring Boot Maven, конфигурация которого была настроена start.spring.io при создании проекта и находится в `pom.xml`. Ниже представлено объявление плагина, обычно размещаемое в конце файла, между тегами `<build><plugins>` и `</plugins></build>`. Он отвечает за дополнительные части стандартной конфигурации, которые вы заметите в проекте:

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

Зависимости Maven, добавляемые start.spring.io при создании проекта

В `pom.xml` вы также обнаружите зависимость, добавленную при создании проекта в start.spring.io — Spring Web. Она подключается так, как показано в следующем фрагменте кода. Это диспетчер зависимостей, называемый `spring-boot-starter-web` (мы подробно рассмотрим диспетчеры зависимостей в подразделе 7.2.2). Версию здесь указывать не нужно.

Во всех созданных нами примерах мы указывали версию каждой зависимости. Здесь это не понадобится, поскольку Spring Boot выбирает нужную версию автоматически. Именно для этого в `pom.xml` есть родительский узел Spring Boot:

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

Файл свойств приложения

И последний важный элемент, который Spring Initializr добавляет в проект, — это файл `application.properties`. Данный файл находится в папке ресурсов проекта Maven. Вначале он пуст и в нашем примере таким и останется. Впоследствии мы поговорим об использовании этого файла для настройки параметров, необходимых в процессе выполнения приложения.

7.2.2. Упрощенное управление зависимостями с помощью диспетчеров зависимостей

Теперь, когда вы научились использовать сервис инициализации проектов и знаете, что собой представляет созданный проект Spring Boot, обратим внимание на следующее важное преимущество изучаемого инструмента — *диспетчеры зависимостей*. Это бесценная функция Spring Boot, способная сэкономить массу времени.

Диспетчер зависимостей — это группа зависимостей, добавленная в конфигурацию приложения с определенной целью. В следующем фрагменте кода вы увидите, что в `pom.xml` диспетчер выглядит как обычная зависимость: его имя, как правило, начинается со `spring-boot-starter-`, после чего стоит оборот, описывающий возможности, которые добавляются в приложение:

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

Предположим, мы хотим добавить в приложение веб-функции. Прежде для описания конфигурации веб-приложения на основе Spring приходилось самостоятельно добавлять в `pom.xml` все необходимые зависимости и следить, чтобы их версии были совместимы между собой. Настраивать их было непросто, а следить за их совместимостью — еще сложнее.

Благодаря диспетчерам больше не приходится обращаться к зависимостям напрямую. Теперь мы запрашиваем функционал (рис. 7.14). Достаточно просто добавить диспетчер зависимости для определенных свойств приложения — например, подключить веб-функции, базу данных, настройки безопасности. Spring Boot автоматически добавит в приложение необходимые зависимости с правильными, совместимыми версиями, которые обеспечат реализацию нужного функционала. Можно сказать, что диспетчеры зависимостей — это группы совместимых зависимостей, ориентированные на предоставление определенного функционала.

Посмотрите на файл `pom.xml`: мы добавили только зависимость `spring-boot-starter-web`; мы не подключали ни контекст Spring, ни AOP, ни Tomcat! Но в папке `External Libraries` нашего приложения вы найдете JAR-архивы и того, и другого, и третьего. Spring Boot знал, что вам понадобится, и загрузил все это, выбрав заведомо совместимые версии.

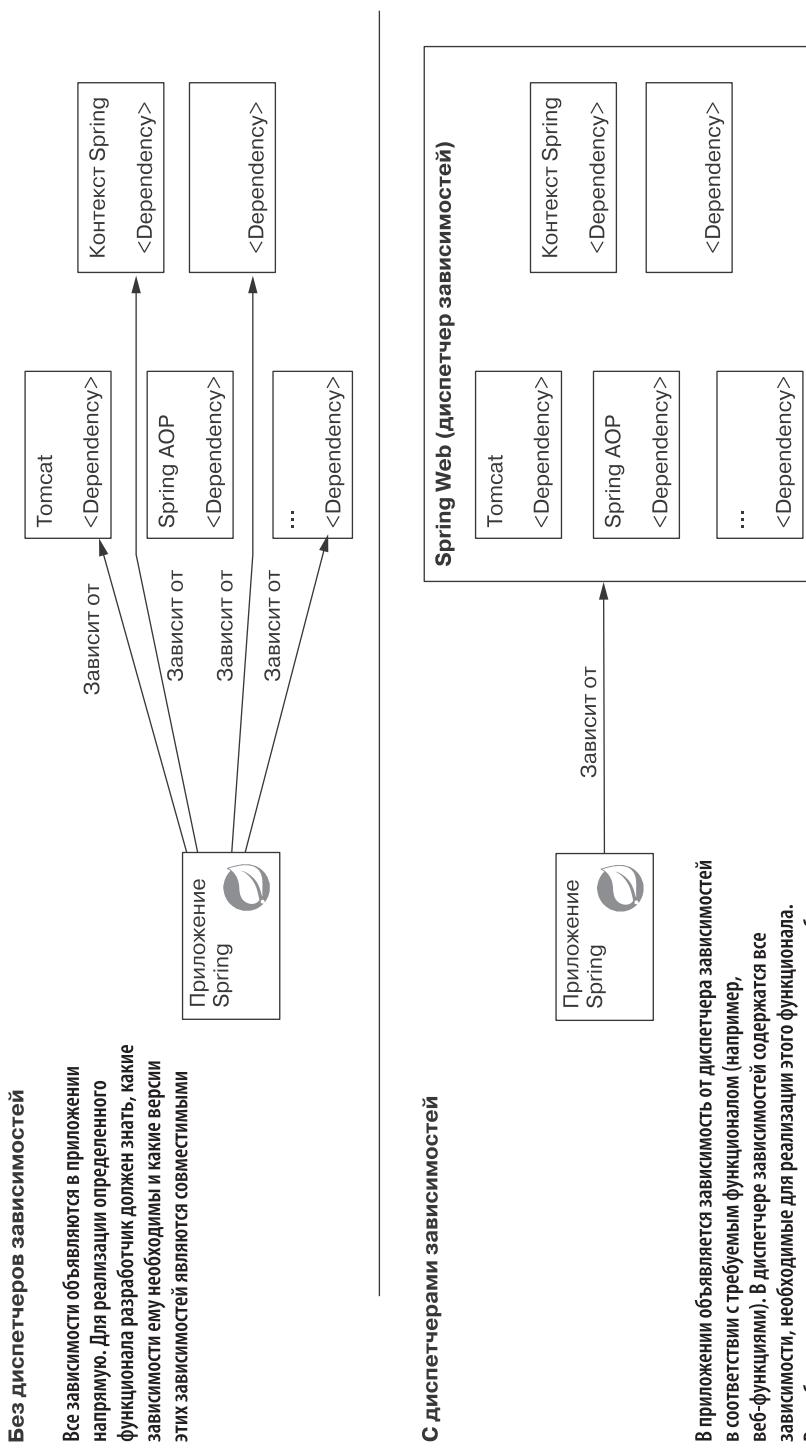


Рис. 7.14. Использование диспетчера зависимостей. Вместо того чтобы ссыльаться на каждую зависимость в отдельности, теперь приложение зависит только от диспетчера. В диспетчере есть все зависимости, необходимые для реализации определенного функционала. Диспетчер также обеспечивает совместимость зависимостей между собой

7.2.3. Автоматическая конфигурация по соглашению на основе зависимостей

Spring Boot также выполняет автоматическую конфигурацию приложения. Принято говорить, что в Spring Boot используется принцип «*соглашения важнее конфигурации*». Далее мы рассмотрим, в чем он заключается и чем может быть полезен.

Автоматическая конфигурация является, пожалуй, самым ценным и популярным из всех описанных в данной главе свойств Spring Boot. Причина этого станет ясна, стоит лишь запустить приложение. Да, я помню, что мы еще ничего не написали, а только загрузили проект и открыли его в IDE. Но мы уже можем запустить это приложение и увидеть, что оно загружает экземпляр Tomcat, который по умолчанию доступен через порт 8080. В консоли вы увидите примерно следующее:

```
Tomcat started on port(s): 8080 (http) with context path ''  
Started Main in 1.684 seconds (JVM running for 2.306)  
| Spring Boot настроил Tomcat и по умолчанию  
| запускает его через порт 8080
```

В соответствии с добавленными зависимостями Spring Boot «догадывается», чего вы ожидаете от приложения, и предоставляет некоторые конфигурации по умолчанию. Это настройки, которые обычно используются для того функционала, для которого вы внедряли зависимости.

Например, когда вы добавили зависимость для веб-функционала, Spring понял, что вам понадобится контейнер сервлетов, и настроил экземпляр Tomcat — как правило, разработчики используют именно его. Tomcat для Spring Boot является контейнером сервлетов по соглашению.

Соглашение — это наиболее частый вариант конфигурации приложения для выполнения определенных задач. Благодаря тому что Spring Boot описывает конфигурацию приложения по соглашению, разработчику остается только изменить те места, где нужны нестандартные настройки. Благодаря этому приходится писать меньше кода в файле конфигурации (или вообще не приходится его писать).

7.3. РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ С ПОМОЩЬЮ SPRING MVC

В этом разделе мы создадим нашу первую страницу веб-приложения на основе Spring. Конечно, у нас уже есть проект Spring Boot с конфигурацией по умолчанию, но приложение пока просто запускает сервер Tomcat. Это еще не делает наше приложение веб-приложением! Нам все еще нужны страницы, которые можно будет открыть в веб-браузере. Продолжим работу над проектом sq-ch7-ex1 и добавим несколько страниц со статическим веб-содержимым. Внеся данные

изменения, вы научитесь создавать веб-страницы и поймете, что при этом происходит в Spring-приложении.

Чтобы добавить в веб-приложение новую страницу, нужно выполнить следующие операции (рис. 7.15).

1. Написать HTML-документ, содержимое которого будет отображаться в браузере.
2. Написать для веб-страницы, созданной в пункте 1, контроллер с соответствующим действием.

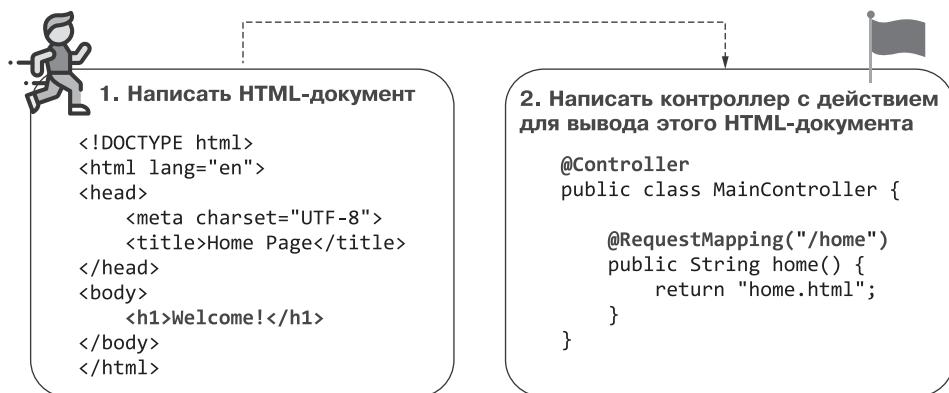


Рис. 7.15. Чтобы добавить в приложение статическую веб-страницу, нужно выполнить следующие две операции: создать HTML-документ с информацией, которую будет выводить браузер, и написать контроллер с действием, назначенным этой странице

В проекте sq-ch7-ex1 мы вначале создадим статическую веб-страницу, содержимое которой должно отображаться в браузере. Это обычный HTML-документ, и в нашем примере он состоит только из короткого заголовка. Содержимое файла с HTML-страницей показано в листинге 7.1. Этот файл нужно поместить в папку `resources/static` проекта Maven. В ней по умолчанию размещаются страницы для визуализации в Spring Boot.

Листинг 7.1. Содержимое HTML-файла

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Home Page</title>
</head>
<body>
    <h1>Welcome!</h1> ← В обычном HTML-документе здесь размещается текст заголовка
</body>
</html>
```

Затем нужно написать контроллер с методом, который будет связывать HTTP-запрос со страницей, возвращаемой приложением в ответ. Контроллер — это компонент веб-приложения, который содержит методы (обычно называемые действиями), выполняемыми для определенных HTTP-запросов. В итоге действие контроллера предоставляет ссылку на веб-страницу, которую приложение дает в ответ на запрос. Чтобы не усложнять первый пример, наш контроллер пока что не будет выполнять какую-то специальную логику. Мы просто настроим действие, которое будет возвращать содержимое документа `home.html`, который мы заранее создали и поместили в папку `resources/static`.

Чтобы отметить класс как контроллер, достаточно воспользоваться аннотацией `@Controller` — это стереотипная аннотация, подобная описанным в главе 4 `@Component` и `@Service`. Таким образом, Spring создаст бин данного класса и добавит его в контекст, чтобы впоследствии этим бином управлять. В классе контроллера можно определить действия контроллера — методы, связанные с определенными HTTP-запросами.

Предположим, мы хотим, чтобы браузер выводил содержимое страницы, когда пользователь задает путь `/home`. Для этого нужно снабдить метод аннотацией `@RequestMapping` и указать данный путь в значении аннотации: `@RequestMapping("/home")`. Метод должен предоставлять строку с именем документа, который приложение будет возвращать в ответ на запрос. Класс контроллера и действие, которое он реализует, представлены в листинге 7.2.

Листинг 7.2. Определение класса контроллера

```
@Controller // Класс сопровождается стереотипной
public class MainController { // аннотацией @Controller

    @RequestMapping("/home") // С помощью аннотации @RequestMapping
    public String home() { // мы привязываем действие к пути HTTP-запроса
        return "home.html"; // Возвращаем имя HTML-документа, в котором содержится то,
    } // что браузер должен вывести на экран
}
```

У вас наверняка уже скопилось множество вопросов! Так случается со всеми моими студентами, когда мы доходим до этого места в изучении Spring. Обычно вопросы такие.

1. Может ли данный метод делать что-нибудь еще, кроме возвращения имени HTML-файла?
2. Может ли этот метод принимать параметры?
3. Помимо `@RequestMapping`, мне встречались примеры с другими аннотациями — чем они лучше?
4. Может ли HTML-страница быть динамической?

На все перечисленные вопросы мы найдем ответы в главе 8. А пока что прошу вас сконцентрироваться на этом простом приложении, чтобы понять, что мы сейчас написали. Прежде всего, следует разобраться, каким образом Spring управляет запросами и вызывает созданное нами действие контроллера. Правильное понимание этого — ценный навык, который впоследствии поможет вам быстрее освоить нюансы работы Spring и, как результат, задавать любые функции веб-приложений.

Теперь запустим веб-приложение, проанализируем его поведение и изучим (в том числе на рисунках) механизм, благодаря которому приложение может так работать. Сначала вы увидите запись в консоли. Там будет сказано, что запускается Tomcat, и будет указан используемый им порт. Если это порт по умолчанию (вы не меняли в конфигурации ничего такого, о чем бы не говорилось в этой главе), то Tomcat будет использовать порт 8080:

```
Tomcat started on port(s): 8080 (http) with context path ''
```

Откройте окно браузера на том же компьютере, где вы запустили приложение, и введите в адресной строке следующий адрес: <http://localhost:8080/home> (рис. 7.16). Не забудьте написать путь /home, который мы связали с действием контроллера, иначе получите ошибку и HTTP-ответ со статусом 404 — Not Found.

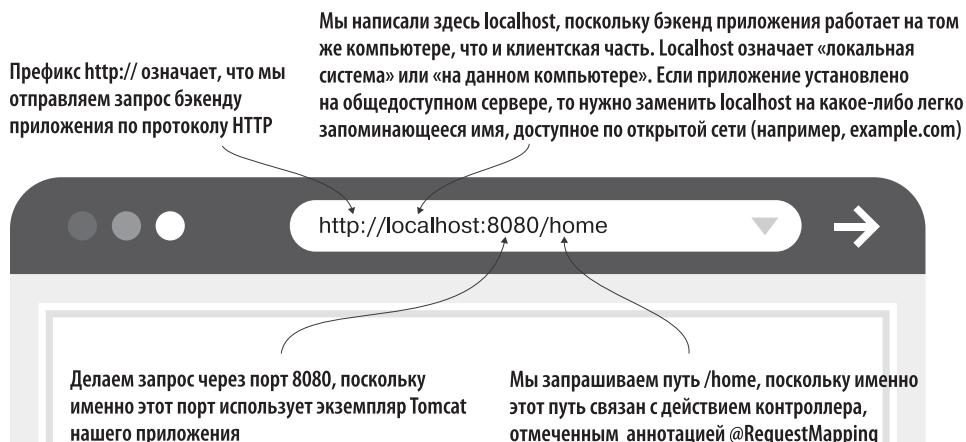


Рис. 7.16. Тестирование приложения. Используя браузер, отправьте запрос бэкенду приложения. Для этого нужно использовать путь, указанный в аннотации `@RequestMapping`. Порт Tomcat должен быть открыт.

На рис. 7.17 показан результат обращения к веб-странице в браузере.

Теперь, когда мы проследили за внешним поведением приложения, рассмотрим, какие механизмы за этим стоят. В Spring есть набор компонентов, которые, взаимодействуя между собой, выдают наблюдаемый нами результат. Данные компоненты и последовательность обработки ими веб-запроса показаны на рис. 7.18.



Welcome!

При обращении к странице, создаваемой приложением, браузер получает содержимое документа home.html. Затем браузер интерпретирует это HTML-содержимое и выводит данные. В случае нашего приложения вы увидите на экране заголовок Welcome!

Рис. 7.17. При обращении к странице в браузере вы увидите заголовок Welcome!. Браузер интерпретирует и выводит HTML-содержимое, полученное от бэкенда в ответ на запрос

- Клиент делает HTTP-запрос.
- Tomcat получает HTTP-запрос клиента и вызывает компонент сервлета, ответственный за его обработку. В случае Spring MVC это сервлет, указанный в конфигурации Spring Boot. Его мы называем *сервлетом-диспетчером* (*dispatcher servlet*).
- Сервлет-диспетчер является точкой входа в веб-приложение Spring. (Именно его мы рассматривали на рис. 7.8 ранее; он же изображен на рис. 7.18.) Tomcat вызывает сервлет-диспетчер для всех полученных HTTP-запросов. Его обязанность — управлять запросами в Spring-приложении. Диспетчер должен найти действие контроллера, которое вызывается в ответ на данный запрос, и определить, что следует вернуть клиенту. Этот сервлет еще называют фронтальным контроллером или единой *точкой входа* (*front controller*).
- Сервлет-диспетчер ищет действие контроллера для ответа на запрос. Чтобы определить, какой контроллер нужно вызвать, сервлет делегирует управление компоненту, который называется *картой обработчиков* (*handler mapping*). Она находит действие контроллера, связанное с запросом через аннотацию `@RequestMapping`.
- Когда нужное действие контроллера будет найдено, сервлет-диспетчер вызывает его. Если на карте обработчиков не нашлось ни одного действия, связанного с запросом, приложение выдает клиенту HTTP-статус 404 — Not Found. Если же действие есть, контроллер возвращает сервлету-диспетчеру имя страницы, которую нужно отобразить. Ее принято называть *представлением* (*view*).
- В этот момент сервлет-диспетчер должен найти представление, имя которого было предоставлено контроллером, получить содержимое представления и отправить его клиенту. Сервлет-диспетчер делегирует обязанность получить содержимое компоненту, который называется *арбитром представлений* (*view resolver*).
- Сервлет-диспетчер возвращает сгенерированное представление клиенту в виде HTTP-ответа.

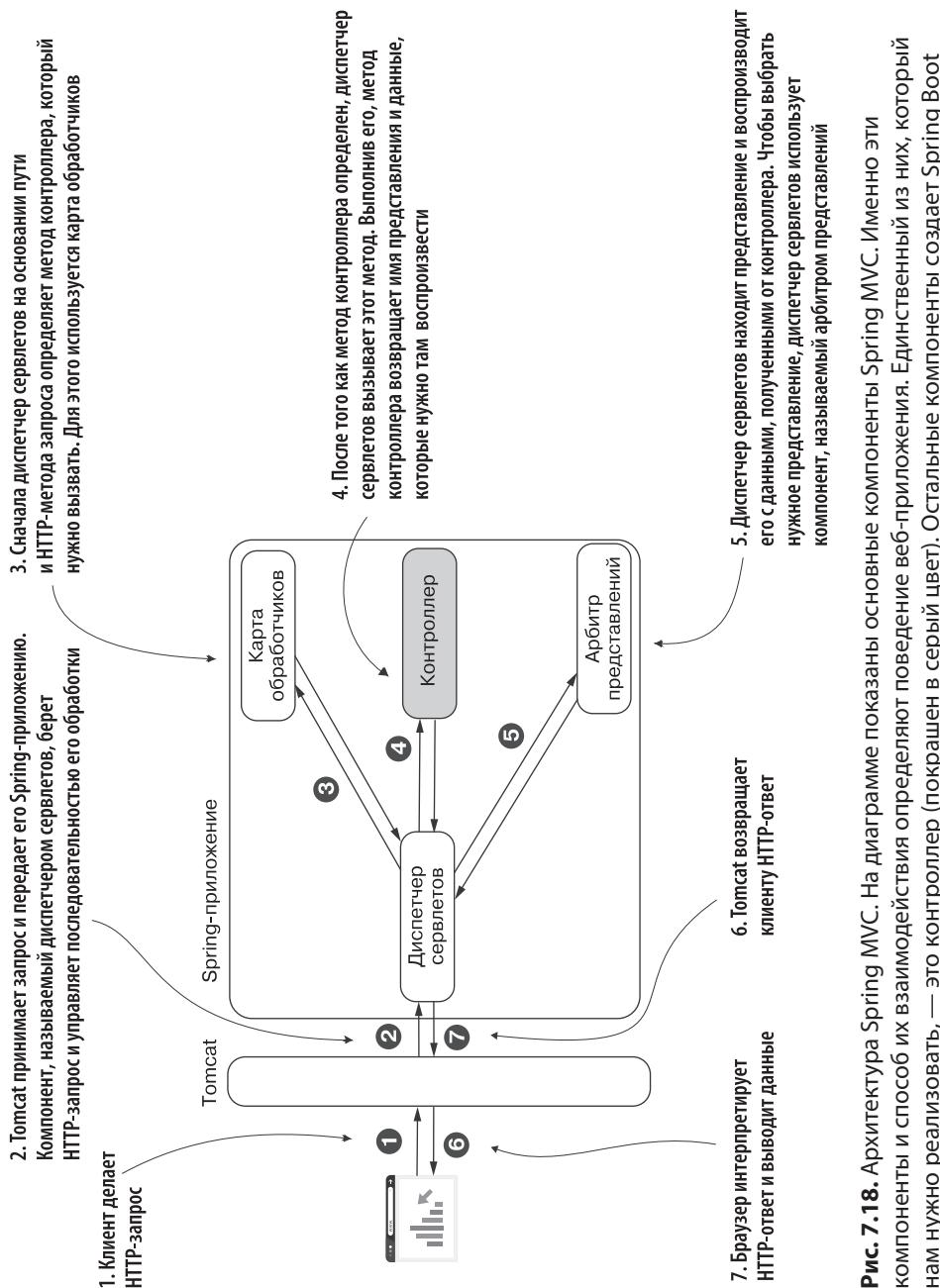


Рис. 7.18. Архитектура Spring MVC. На диаграмме показаны основные компоненты Spring MVC. Именно эти компоненты и способ их взаимодействия определяют поведение веб-приложения. Единственный из них, который нам нужно реализовать, — это контроллер (покрашен в серый цвет). Остальные компоненты создает Spring Boot

ПРИМЕЧАНИЕ

В этой главе карта обработчиков описывается как компонент, который находит нужное действие контроллера для заданного пути HTTP-запроса. Кроме этого, карта обработчиков также ищет нечто называемое HTTP-методом, но я пока не затрагиваю данную тему, чтобы вам было проще сконцентрироваться на последовательности обработки HTTP-запроса. Мы подробно рассмотрим HTTP-методы в главе 8.

Благодаря этим настройкам Spring (в сочетании со Spring Boot) сильно упрощает разработку веб-приложений. Вам остается только написать действия контроллера и связать их с запросами посредством аннотаций. Значительная часть логики скрыта внутри фреймворка, что ускоряет написание приложений и сокращает их код.

В главе 8 мы более подробно рассмотрим возможности класса контроллера. Реальные приложения выполняют гораздо более сложные вещи, чем просто возвращение содержимого статической HTML-страницы. Как правило, страница содержит динамические элементы, обработанные приложением перед воспроизведением HTTP-ответа. Но пока что на минутку остановимся и вспомним, что мы изучили в этой главе. Понимание основных принципов работы веб-приложений на базе Spring необходимо для освоения последующих глав. И уж конечно, без него вы не станете профессиональным разработчиком Spring-приложений. «Не углубляйся в детали, прежде чем как следует не поймешь основы» — это главное правило, которым я пользуюсь при изучении любой технологии.

РЕЗЮМЕ

- Сегодня веб-приложения используют гораздо чаще, чем десктопные программы. Поэтому необходимо изучить принципы работы веб-приложений и научиться их создавать.
- Веб-приложение — это приложение, с которым пользователь взаимодействует посредством веб-браузера. Оно имеет клиентскую и серверную части. Серверная часть (бэкенд) отвечает за обработку и хранение данных. Клиентская часть (фронтенд) отправляет запросы к серверной части. Бэкенд выполняет действия, соответствующие запросу фронтенда, и возвращает фронтенду ответ.
- Spring предоставляет функционал для реализации веб-приложений. Чтобы не писать много строк конфигурации, можно воспользоваться Spring Boot — проектом экосистемы Spring, который, действуя по принципу «соглашения важнее конфигурации», предоставляет настройки по умолчанию для всех необходимых функций приложения.

- Spring Boot также упрощает конфигурацию зависимостей, поскольку использует диспетчер зависимостей. Диспетчер зависимостей — это группа зависимостей, имеющих совместимые версии и обеспечивающих определенный функционал приложения.
- Для получения HTTP-запросов и передачи ответов в бэкенде веб-приложения Java используется контейнер сервлетов (такой как Tomcat) — программное обеспечение, способное преобразовывать HTTP-запросы и ответы в данные, воспринимаемые Java-приложением. Благодаря контейнеру сервлетов разработчику не нужно писать код для обмена данными по сети с использованием HTTP-протокола.
- Проект Spring Boot значительно упрощает создание веб-приложения, так как автоматически генерирует конфигурацию контейнера сервлетов. Spring Boot также обладает необходимым функционалом для написания сценариев использования продукта. Кроме того, Spring Boot самостоятельно создает конфигурацию для набора компонентов, которые перехватывают HTTP-запросы и управляют ими. Эти компоненты являются частью структуры классов, именуемой Spring MVC.
- Поскольку Spring Boot автоматически генерирует конфигурацию компонентов Spring MVC и контейнера сервлетов, для организации простейшего обмена запросами и ответами HTTP разработчику остается только написать HTML-документ, который веб-приложение передаст клиенту, и класс контроллера.
- Для описания конфигурации контроллера и его действий используются аннотации. Чтобы отметить класс как контроллер Spring MVC, применяется стереотипная аннотация `@Controller`, а чтобы связать действие контроллера с определенным HTTP-запросом — аннотация `@RequestMapping`.

Реализация веб-приложений с использованием Spring Boot и Spring MVC

В этой главе

- ✓ Применение шаблонизаторов для построения динамических представлений.
- ✓ Передача данных от клиента к серверу посредством HTTP-запросов.
- ✓ Создание HTTP-запросов с использованием HTTP-методов GET и POST.

В главе 7 вы получили общее представление о том, как использовать Spring при создании веб-приложений. Мы рассмотрели компоненты веб-приложений и зависимости, в которых такие приложения нуждаются, вы также познакомились с архитектурой Spring MVC. Мы даже написали первое веб-приложение, чтобы пронаблюдать за совместной работой всех этих компонентов.

Теперь мы пойдем дальше и добавим в проект несколько функций, которые часто встречаются в современных приложениях. Сначала мы создадим несколько страниц, содержимое которых изменяется в зависимости от того, как приложение обрабатывает данные конкретного запроса. В наше время статические страницы встречаются очень редко. И если вы сейчас подумали, что должен быть способ выбрать контент для страницы перед тем, как отправить браузеру HTTP-ответ, то вы правы: есть даже несколько вариантов для этого!

В разделе 8.1 мы будем создавать динамические представления на основе шаблонизаторов. Шаблонизатор — это зависимость, которая упрощает получение

и вывод изменяемых данных от контроллера. Мы познакомимся с процессом Spring MVC, а затем рассмотрим работу шаблонизатора на конкретном примере.

В разделе 8.2 вы научитесь передавать данные от клиента серверу посредством HTTP-запросов. Мы будем использовать эти данные в методе контроллера и формировать представления с динамическим содержимым.

В разделе 8.3 мы рассмотрим HTTP-методы. Вы узнаете, что путь — это еще не все, что нужно для идентификации запроса клиента. Кроме него, клиент использует один из HTTP-методов (GET, POST, PUT, DELETE, PATCH и т. д.), отражающий его намерения. В качестве примера мы создадим HTML-форму, которую можно применять для передачи бэкенду значений на обработку. Впоследствии, в главах 12 и 13, вы научитесь сохранять эти данные в базе данных — таким образом, ваши приложения постепенно будут становиться все ближе к реальным готовым продуктам.

8.1. СОЗДАНИЕ ВЕБ-ПРИЛОЖЕНИЙ С ДИНАМИЧЕСКИМИ ПРЕДСТАВЛЕНИЯМИ

Предположим, нам нужно создать страницу корзины для интернет-магазина. Эта страница не может показывать одну и ту же информацию всем покупателям. И даже один и тот же посетитель не будет видеть там одно и то же. На странице должны каждый раз выводиться только товары, которые данный пользователь добавил в корзину в этот раз. На рис. 8.1 показан пример динамического представления — это корзина покупок на сайте Manning. Посмотрите, как по запросу к Manning.com/cart каждый раз выводятся разные данные: страница одна и та же, а информация на ней всякий раз другая. Все дело в том, что у нее динамическое содержимое!

Далее мы создадим веб-приложение с динамическим представлением. Большинство современных приложений предоставляют пользователю динамические данные. В этом случае, получив запрос пользователя, выраженный в форме переданного браузером HTTP-запроса, веб-приложение получает некие данные, обрабатывает их и возвращает HTTP-ответ, который браузер выводит на экран (рис. 8.2).

Мы кратко рассмотрим процесс обработки запросов Spring MVC и затем на практике попробуем разобраться, как именно представление получает динамические значения от контроллера.

В примере, который мы рассмотрели в главе 7, в ответ на все HTTP-запросы браузер выводил одно и то же содержимое страницы. Напомню, что процесс обработки запросов Spring MVC выглядит следующим образом (рис. 8.3).

1. Клиент отправляет на сервер HTTP-запрос.
2. Диспетчер сервлетов с помощью карты обработчиков находит нужный контроллер и действие, которое необходимо выполнить.

218 Часть II. Реализация

3. Диспетчер вызывает это действие.
4. После того как действие будет выполнено, контроллер возвращает имя представления, которое диспетчер сервлетов должен преобразовать в HTTP-ответ.
5. HTTP-ответ возвращается клиенту.

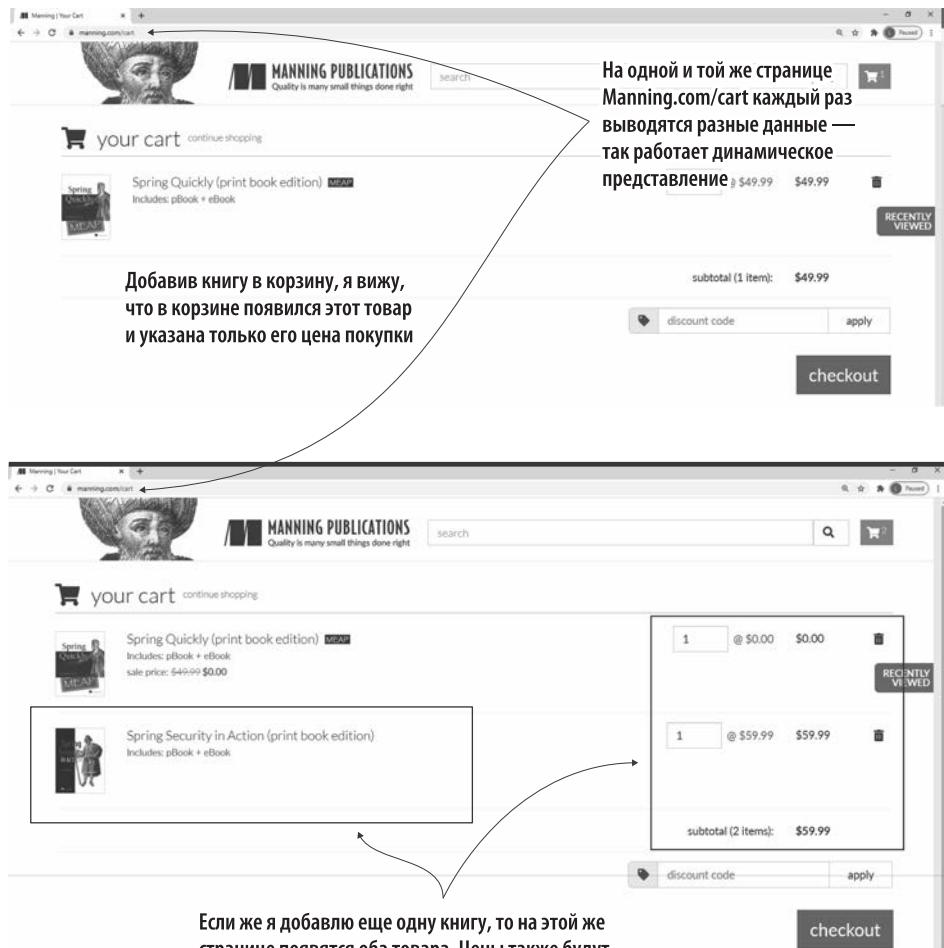


Рис. 8.1. Динамическое представление для корзины интернет-магазина Manning. Несмотря на то что при каждом запросе указывается один и тот же путь, содержимое страницы постоянно изменяется. В ответ перед добавлением товара в корзину и после него бэкенд отправляет разные данные

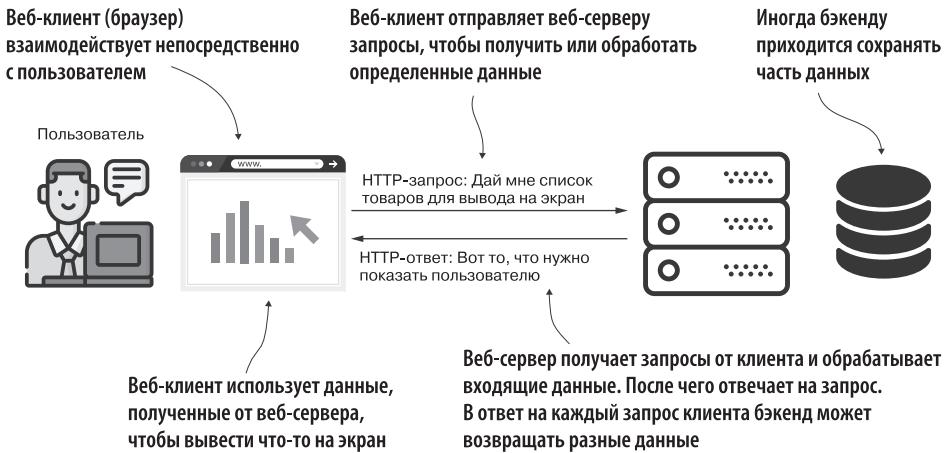


Рис. 8.2. Клиент отправляет данные посредством HTTP-запроса. Бэкенд их обрабатывает и формирует ответ, который возвращает клиенту. В зависимости от того, каким образом бэкенд работает с данными, результат клиент будет получать разный

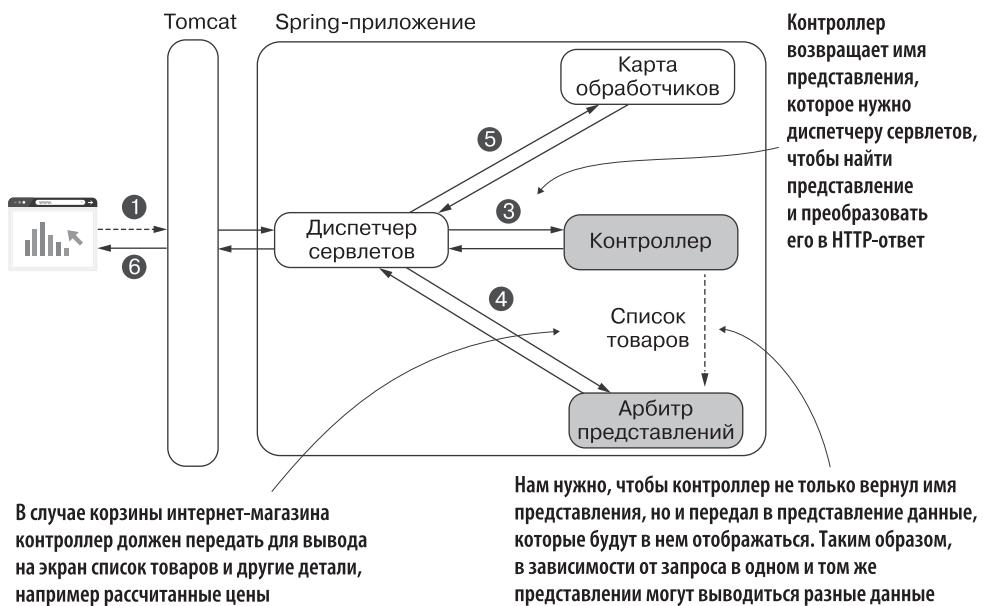


Рис. 8.3. Процесс обработки запросов Spring MVC. Чтобы сформировать динамическое представление, контроллер должен передать для него данные. Эти данные могут различаться в зависимости от запроса. Например, для корзины интернет-магазина контроллер вначале предоставляет список, состоящий из одного товара. После того как пользователь добавляет в корзину еще несколько продуктов, контроллер включает в перечень и их. Таким образом, одно и то же представление дает разную информацию в ответ на эти запросы

Здесь нужно внести изменения в пункт 4. Мы хотим, чтобы контроллер не только возвращал имя представления, но и каким-то образом передавал в представление данные для их вставки в HTTP-ответ. Таким образом, если сервер предоставит список с одним товаром, на странице, отображающей данный перечень, будет выведена только эта позиция. Но если потом контроллер передаст для этого же представления два товара, данные на экране изменятся — количество продуктов в корзине удвоится (именно такое поведение мы наблюдали на рис. 8.1).

Теперь я покажу на простом примере, каким образом данные из контроллера передаются в представление. Этот пример вы найдете в проекте sq-ch8-ex1. В нем мы сосредоточимся на синтаксисе. Но впоследствии вы сможете использовать изученный прием для передачи любых данных из контроллера в представление.

Предположим, мы хотим передать имя и выделить его определенным цветом. На практике часто приходится отображать на странице имя пользователя. Как это сделать? Как получить данные, которые могут различаться для разных запросов, и вывести их?

Создадим проект Spring Boot (sq-ch8-ex1) и добавим шаблонизатор в раздел зависимостей файла pom.xml. Мы будем использовать шаблонизатор, который называется Thymeleaf, — это зависимость, позволяющая легко передавать данные из контроллера в представление и выводить их определенным образом. Мне нравится Thymeleaf, поскольку он проще других аналогичных инструментов и, на мой взгляд, его проще понять и освоить. Как вы убедитесь на нашем примере, шаблоны Thymeleaf — это просто статические HTML-файлы. В следующем фрагменте кода показано, какую зависимость нужно добавить в pom.xml:

```
<dependency>                                            Диспетчер зависимостей, который добавляется
    <groupId>org.springframework.boot</groupId>    для подключения шаблонизатора Thymeleaf
    <artifactId>spring-boot-starter-thymeleaf</artifactId> ←
</dependency>
<dependency>                                            Поскольку мы создаем веб-приложение,
    <groupId>org.springframework.boot</groupId>    нам также нужно добавить диспетчер
    <artifactId>spring-boot-starter-web</artifactId> ←  зависимостей для веб-приложений
</dependency>
```

В листинге 8.1 представлено определение контроллера. Как мы уже знаем из главы 7, метод, связывающий действие с путем запроса, сопровождается аннотацией @RequestMapping. Теперь мы определим для этого метода параметр типа Model, где хранятся данные, которые контроллер будет передавать в представление. В экземпляр Model мы добавим значения, которые также должны отправиться в представление. Каждое из них будет иметь уникальное имя (еще называемое ключом). Чтобы добавить новое значение, которое контроллер будет передавать в представление, мы будем вызывать метод addAttribute(). Первым параметром метода addAttribute() является ключ, вторым — значение для представления.

Листинг 8.1. Класс контроллера, в котором определено действие для страницы

```

@Controller           ← Стереотипная аннотация @Controller говорит о том, что перед нами класс
public class MainController {           контроллера Spring MVC, и добавляет в контекст Spring бин данного типа

    @RequestMapping("/home")           ← Назначаем действию контроллера путь HTTP-запроса
    public String home(Model page) {   ← Метод действия определяет параметр
        page.addAttribute("username", "Katy");   ← типа Model, в котором хранятся
        page.addAttribute("color", "red");       ← данные, передаваемые контроллером
        return "home.html";                ← в представление
    }
}                           ← Добавляем данные, которые контроллер
                           ← должен отправить в представление
                           ← Действие контроллера возвращает представление,
                           ← которое затем будет преобразовано в HTTP-ответ
}

```

ПРИМЕЧАНИЕ

Студенты иногда спрашивают меня, почему, если ввести в адресной строке браузера просто localhost:8080, без указания пути вроде /home, возникает ошибка. А она появляется совершенно законно. Страница с сообщением об ошибке выводится приложением Spring Boot, когда то получает HTTP-ответ в виде статуса HTTP 404 (Not Found).

Вводя в браузере просто localhost:8080, мы обращаемся к серверу по пути "/". Поскольку с этим путем не связано ни одно действие контроллера, сервер выдает HTTP-статус 404. Если вы планировали получить вместо этого что-то другое, назначьте действие контроллера для данного пути, снова воспользовавшись аннотацией @RequestMapping.

Чтобы определить представление, нужно поместить в папку resources/templates проекта Spring Boot файл home.html. Обратите внимание на небольшое различие: в главе 7 мы отправили HTML-файл в папку resources/static, так как создавали статическое представление. Теперь, поскольку мы используем шаблонизатор и работаем над динамическим представлением, нам нужно разместить HTML-файл в папке resources/templates.

Содержимое файла home.html, которое я добавил в проект, представлено в листинге 8.2. Первый важный момент, на который нужно обратить внимание, — это тег <html>, с которого начинается содержимое файла и в который я добавил атрибут xmlns:th="http://www.thymeleaf.org". Данное определение эквивалентно импорту в Java. Впоследствии оно позволит нам обозначать префиксом th все специальные свойства представления, полученные благодаря Thymeleaf. Немного дальше вам встретится два фрагмента, где я отметил префиксом th данные, передаваемые в приложение контроллером. Используя синтаксис \${attribute_key}, можно обозначить любые атрибуты, отправленные из контроллера через экземпляр Model. Например, я использовал конструкцию \${username}, чтобы получить значение атрибута username, и \${color} — для атрибута color.

Чтобы убедиться, что все это работает, запустите приложение и откройте веб-страницу в браузере. Она будет выглядеть примерно так, как на рис. 8.4.

Листинг 8.2. Файл home.html с динамическим представлением приложения

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"> ← Определяем префикс th
    <head>
        <meta charset="UTF-8">
        <title>Home Page</title>
    </head>

    <body>
        <h1>Welcome
            <span th:style="color: ' + ${color}" th:text="${username}"></span>!</h1> | Используем префикс th для значений,
        переданных контроллером
    </body>

</html>

```



Welcome Katy!

Имя, которое выводится на странице, — это значение, полученное от контроллера. Оно выводится красным цветом, так как значение цвета также было передано контроллером

Рис. 8.4. Результат. Если запустить приложение и открыть страницу в браузере, увидим, что в представлении были использованы значения, переданные контроллером

Теперь в представлении используются все данные, передаваемые контроллером.

8.1.1. Получение данных из HTTP-запроса

Обсудим, как клиент передает данные серверу посредством HTTP-запросов. В приложениях часто клиенту нужна такая возможность. Полученные сервером данные обрабатываются и затем выводятся в представлении, как было показано в разделе 8.1. Вот несколько сценариев использования, где клиенту необходимо передавать данные серверу:

- реализация заказов в интернет-магазине: клиент должен передать на сервер список товаров, заказанных пользователем. Затем сервер обрабатывает заказ;
- веб-форум, на котором пользователи могут создавать и редактировать посты: клиент передает на сервер сведения о посте. Сервер сохраняет эту информацию в базе данных или изменяет уже имеющуюся там;

- регистрация в приложении. Пользователи вводят свои учетные данные, которые затем нужно валидировать. Клиент передает информацию на сервер, а сервер ее верифицирует;
- страница контактных данных, где размещена форма, в которой пользователь может написать тему и текст сообщения, а затем отослать его по определенному адресу электронной почты. Клиент передает эти значения на сервер, сервер обрабатывает их и отправляет электронное письмо по заданному пути.

В большинстве случаев данные передаются посредством HTTP-запроса одним из следующих способов:

- *в виде параметра HTTP-запроса* — это простой способ передачи значений от клиента к серверу в формате пары «ключ — значение». Параметры добавляются в URI как выражения запроса. Поэтому их также называют *параметрами запроса*. Этот метод следует использовать только для передачи небольших объемов данных;
- *в виде параметров заголовка HTTP-запроса*. Как и параметры запроса, параметры заголовка передаются в заголовке HTTP-запроса. Главное различие между ними — параметры заголовка не попадают в URI. Этот способ также не подходит для больших объемов данных;
- *переменная пути* передает данные через сам путь запроса. Как и в случаях выше, переменные пути используются для небольших объемов данных. Но данный вариант передачи следует применять, если данные являются обязательными;
- *в теле HTTP-запроса*. Этот метод обычно применяется в случаях, когда нужно передать много данных (строки, но иногда и двоичные данные, такие как файлы). Мы рассмотрим его в главе 10, когда научимся создавать конечные точки REST.

8.1.2. Передача данных от клиента серверу посредством параметров запроса

Ниже мы выполним пример, демонстрирующий использование параметров HTTP-запроса — простой способ передачи данных от клиента бэкенду. Этот прием часто встречается в реальных приложениях. Параметры запроса применяют в следующих случаях:

- *объем данных не очень велик*. Параметры запроса задаются посредством переменных запроса (как будет показано далее). Этот метод позволяет отправить максимум около 2000 символов;
- *нужно передать необязательные данные*. Параметры запроса — простой способ выслать значения, которые клиент не обязан передавать. Сервер будет готов не получить значения определенных параметров запроса.

Типичный случай использования параметров запроса — определение критериев поиска и фильтрации (рис. 8.5). Предположим, приложение выводит товары и их свойства в виде таблицы. У каждого есть наименование, цена и бренд. Мы хотим, чтобы пользователь мог найти товар по любому из этих критериев: цене, наименованию, бренду или по сочетанию данных характеристик. Здесь было бы правильно использовать параметры запроса. Приложение отправляет значения (наименование, цену, бренд) в виде необязательных параметров. Клиенту нужно передать только те из них, по которым пользователь решил выполнить поиск.



Рис. 8.5. Параметры запроса не являются обязательными. Типичный случай их использования — реализация поиска с опциональными критериями. Клиент передает лишь некоторые параметры запроса, и сервер знает, что нужно использовать только полученные значения. При создании сервера необходимо учитывать, что значения некоторых параметров могут отсутствовать

Модифицируем пример, рассмотренный в начале раздела 8.1, использовав параметры запроса для передачи цвета, которым выводится имя, полученное от клиента. В листинге 8.3 показано, что нужно сделать с классом контроллера, чтобы можно было получить значение цвета, переданное клиентом в качестве параметра запроса. Я выделил этот пример в отдельный проект `sq-ch8-ex2` — так вам будет удобнее следить за изменениями. Чтобы извлечь значение из параметра запроса, нужно добавить в метод действия контроллера еще один параметр

и снабдить его аннотацией `@RequestParam`. Она сообщает Spring, что нужно извлечь значение из параметра HTTP-запроса, причем имя этого параметра совпадает с именем параметра метода.

Листинг 8.3. Получение значения через параметр запроса

```
@Controller
public class MainController {
    @RequestMapping("/home")
    public String home(
        @RequestParam String color,
        Model page) {
        page.addAttribute("username", "Katy");
        page.addAttribute("color", color);
        return "home.html";
    }
}
```

Определяем новый параметр для метода действия контроллера и сопровождаем этот параметр аннотацией `@RequestParam`

Также добавляем параметр типа `Model`, который используется для передачи данных из контроллера в представление

Контроллер передает в представление цвет, полученный от клиента

На рис. 8.6 показано, как значение параметра `color` передается от клиента в действие контроллера в бэкенде, чтобы затем быть использованным в представлении.

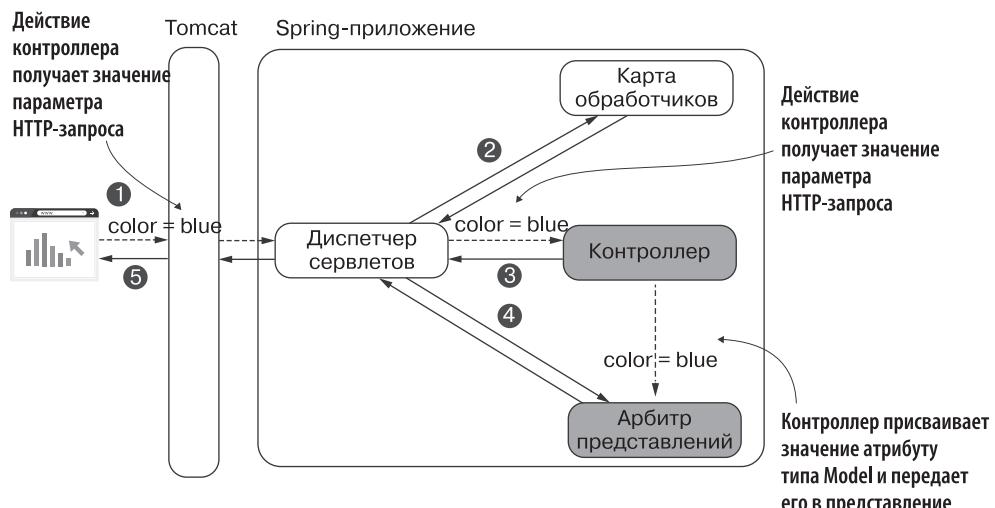


Рис. 8.6. Путь значения, переданного клиентом, по архитектуре Spring MVC. Действие контроллера получает от клиента параметры запроса и может их использовать. В данном примере значение присваивается параметру типа `Model` и передается в представление

Теперь запустите приложение и введите в браузере путь `/home`. Чтобы присвоить значение параметру запроса, используйте следующий синтаксис:

`http://localhost:8080/home?color=blue`

Дабы присвоить значения параметрам HTTP-запроса, нужно поставить в конце пути вопросительный знак (?), после которого перечислить параметры пар в виде «ключ — значение», разделенных символом &. Например, для передачи имени в качестве параметра запроса я написал следующее:

```
http://localhost:8080/home?color=blue&name=Jane
```

Для получения этого параметра также нужно добавить новый параметр в действие контроллера. Измененный код показан в следующем фрагменте. Этот пример вы найдете в проекте sq-ch8-ex3.

```
@Controller
public class MainController {
    @RequestMapping("/home")
    public String home(
        @RequestParam(required = false) String name,
        @RequestParam(required = false) String color,
        Model page) {
        page.addAttribute("username", name); ← Получаем новый параметр запроса name
        page.addAttribute("color", color); ← Передаем значение параметра name в представление
        return "home.html";
    }
}
```

В группе «ключ — значение» (такой как `color=blue`) ключ — это имя параметра запроса, а значение — то, что стоит сразу после знака равенства (=).

Синтаксис параметров запроса наглядно представлен на рис. 8.7.



Рис. 8.7. Передача данных через параметры запроса. Каждый из них представлен в виде пары «ключ — значение». Список параметров входит в состав пути и начинается с вопросительного знака (?). Если их несколько, они разделяются символом &

ПРИМЕЧАНИЕ

По умолчанию параметр запроса является обязательным. Если клиент не передает значение для него, сервер возвращает HTTP-статус 400 Bad Request. Чтобы значение перестало быть обязательным, нужно явно прописать это в аннотации с помощью дополнительного атрибута: `@RequestParam(optional=true)`.

8.1.3. Передача данных от клиента серверу с помощью переменных пути

Теперь рассмотрим передачу данных от клиента серверу с помощью переменных пути и сравним этот способ с тем, что мы изучили выше. Использование переменных пути — еще один вариант отправки данных от клиента серверу. Однако, в отличие от параметров HTTP-запроса, в этом случае мы явно вставляем переменные в путь, как показано далее.

Параметры запроса:

`http://localhost:8080/home?color=blue`

Переменные пути:

`http://localhost:8080/home/blue`

Теперь не приходится указывать ключ для значения. Достаточно вставить значение в определенное место пути — и на стороне сервера оно будет извлечено из этой позиции. В пути может быть несколько значений как переменных пути, но обычно рекомендуется использовать одно-два. Вы сами убедитесь, что с большим количеством переменных путь становится трудно читать. Если значений больше двух, вместо переменных пути я предпоютаю параметры запроса, описанные в подразделе 8.1.2. Не следует также применять переменные пути для optionalных значений. Я советую их использовать только для обязательных параметров. Для необязательных лучше подойдут параметры запроса, описанные в подразделе 8.1.2. В табл. 8.1 приводится сравнение параметров запроса и переменных пути.

Таблица 8.1. Краткое сравнение параметров запроса и переменных пути

Параметры запроса	Переменные пути
1. Могут использоваться для необязательных значений	1. Не могут использоваться для необязательных значений
2. Не рекомендуется использовать много таких параметров. Как вы узнаете из главы 10, если их больше трех, лучше поместить их в тело запроса — хотя бы для удобства чтения	2. Не стоит передавать более трех переменных пути. Лучше даже ограничиться максимум двумя
3. По мнению некоторых разработчиков, выражения запроса труднее читать, чем выражения пути	3. Переменные пути легче читать, чем переменные запроса. В случае общедоступных веб-сайтов страницы с переменными путем легче индексируются поисковыми системами (такими как Google), благодаря чему веб-сайт проще найти через поисковик

Если создаваемая вами страница зависит всего лишь от одного или двух значений, определяющих ее итоговый вид, лучше вставить их непосредственно в путь, чтобы запрос было проще читать. Кроме того, подобный URL будет проще найти, если сохранить его в браузере в виде закладки и он будет легче индексироваться поисковыми системами (если это важно для вашего приложения).

Выполним пример, демонстрирующий синтаксис, который необходимо написать в контроллере, чтобы получить значения переменных пути. Я изменил примеры, с которыми мы работали в подразделе 8.1.2, но выделил этот код в новый проект sq-ch8-ex4, чтобы вам было проще его проверить.

Для отправки переменной пути в действие контроллера достаточно добавить в путь имя этой переменной, заключенное в фигурные скобки (как показано в листинге 8.4). Затем с помощью аннотации `@PathVariable` мы отметим параметр действия контроллера, в котором будет передаваться значение переменной пути. В листинге 8.4 показано, как нужно изменить действие контроллера, чтобы получить значение цвета из переменной пути (остальной код примера не отличается от проекта sq-ch8-ex2, рассмотренного в подразделе 8.1.1).

Листинг 8.4. Передача значений от клиента с помощью переменных пути

```
Controller
public class MainController {
    @RequestMapping("/home/{color}")
    public String home(
        @PathVariable String color, ←
        Model page) { ←
        page.addAttribute("username", "Katy");
        page.addAttribute("color", color);
        return "home.html";
    }
}
```

Чтобы определить переменную пути, нужно присвоить ей имя и вставить это имя в путь, заключив в фигурные скобки

Параметр, который должен принимать переменную пути, нужно отметить аннотацией `@PathVariable`. Имя параметра должно совпадать с именем переменной пути

Запустите приложение и откройте в браузере страницу с разными значениями цвета:

```
http://localhost:8080/home/blue
http://localhost:8080/home/red
http://localhost:8080/home/green
```

В каждом запросе имя выводится на странице своим цветом. На рис. 8.8 показано, как связаны код и путь запроса.

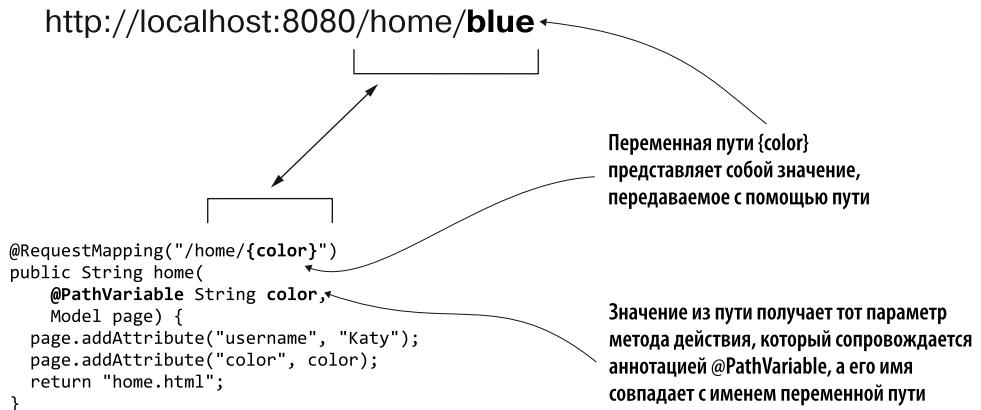


Рис. 8.8. Использование переменных пути. Чтобы извлечь значение из переменной пути, нужно, указывая путь в действии контроллера, заключить имя этой переменной в фигурные скобки. Чтобы получить значение переменной пути, используется параметр с аннотацией `@PathVariable`

8.2. ИСПОЛЬЗОВАНИЕ HTTP-МЕТОДОВ GET И POST

Далее мы рассмотрим HTTP-методы и то, как с их помощью клиент может сообщить, что именно он хочет сделать с запрошенным ресурсом (создать, изменить, получить, удалить). HTTP-запрос состоит из пути и этого действия. До сих пор мы говорили только о пути и, сами того не сознавая, использовали HTTP-метод GET. Назначение HTTP-метода — определить действие, которое запрашивает клиент. Например, метод GET показывает лишь желание извлечь данные. Применяя его, клиент сообщает серверу, что он хочет что-то получить, но никакие данные при этом не должны быть изменены. Однако иногда этого мало. В приложении также часто возникает потребность что-то изменять, создавать и удалять.

ПРИМЕЧАНИЕ

Будьте внимательны! Вы можете использовать HTTP-метод не по его прямому назначению, но так делать некорректно. Например, с помощью метода GET все же можно реализовать функционал, который будет изменять данные. Технически это возможно, но это очень, очень плохое решение. Никогда не используйте HTTP-метод не по назначению.

До сих пор, чтобы обратиться к определенному методу контроллера, мы использовали путь запроса. Но в более сложных случаях один и тот же путь может

быть связан с несколькими действиями контроллера, поскольку используются разные HTTP-методы. Рассмотрим такой вариант на примере.

HTTP-метод определяется ключевым словом, соответствующим намерению клиента. Если запрос клиента требует только получения данных, мы создаем конечную точку, используя HTTP-метод GET. Если же при этом данные на сервере как-то изменяются, для точного описания действий клиента нужно использовать другое ключевое слово.

В табл. 8.2 представлены основные HTTP-методы, которые применяются в приложениях и о которых вам следует знать.

Таблица 8.2. Основные HTTP-методы, часто встречающиеся в веб-приложениях

HTTP-метод	Описание
GET	Клиентский запрос лишь получает данные
POST	Клиентский запрос передает новые данные, которые должны быть сохранены на сервере
PUT	Клиентский запрос изменяет данные, хранящиеся на сервере
PATCH	Клиентский запрос требует частичного изменения данных, хранящихся на сервере
DELETE	Клиентский запрос удаляет данные на сервере

Чтобы вам было проще запомнить назначение этих методов, они наглядно представлены на рис. 8.9.

ПРИМЕЧАНИЕ

Несмотря на то что хорошим тоном программирования считается различать полное (PUT) и частичное (PATCH) изменение данных, при разработке реальных приложений различия между этими HTTP-методами часто не делают.

Теперь рассмотрим пример, в котором, помимо GET, будут использоваться другие HTTP-методы. Предположим, нам нужно создать приложение, которое сохраняет список товаров. У каждого продукта есть наименование и цена. Веб-приложение выводит перечень всех товаров, и пользователь может добавить туда один или несколько продуктов посредством HTML-формы.

Обратите внимание на два сценария использования, описанные в данной задаче. Пользователю нужно дать возможность сделать следующее:

- просмотреть список всех товаров; здесь мы по-прежнему будем использовать HTTP-метод GET;
- добавить товары в список: здесь мы будем использовать HTTP-метод POST.

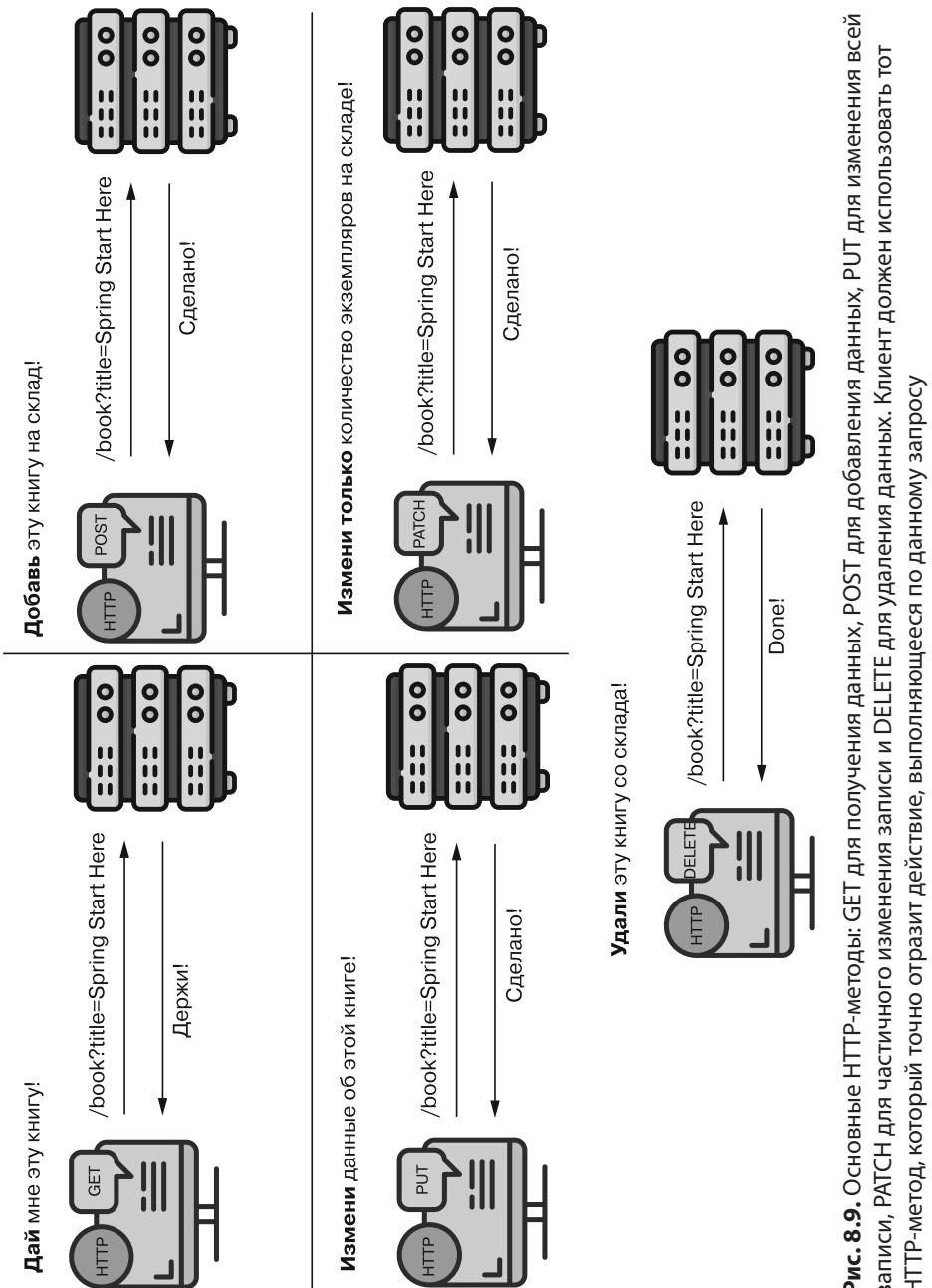


Рис. 8.9. Основные HTTP-методы: GET для получения данных, POST для добавления данных, PUT для изменения всей записи, PATCH для частичного изменения записи и DELETE для удаления данных. Клиент должен использовать тот HTTP-метод, который точно отразит действие, выполняющееся по данному запросу

Создадим новый проект sq-ch8-ex5 с зависимостями (в файле pom.xml) для веб-приложений и Thymeleaf:

```
<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>
```

В этом проекте мы создадим класс `Product`, в котором описывается товар, с атрибутами для наименования и цены. Как мы уже знаем из главы 5, класс `Product` — это класс модели, так что мы поместим его в пакет `model`. Класс `Product` представлен в листинге 8.5.

Листинг 8.5. Класс `Product`, описывающий товар, с атрибутами наименования и цены

```
public class Product {

    private String name;
    private double price;

    // геттеры и сеттеры
}
```

Теперь, когда у нас есть способ описания товаров, создадим список, где приложение будет их сохранять. Веб-приложение будет выводить перечень продуктов, и пользователь сможет добавлять в него новые наименования. Мы реализуем два сценария использования (получение списка товаров для отображения на экране и добавление нового товара) в виде метода класса сервиса. Создадим класс сервиса `ProductService` и поместим его в пакет `service`.

В листинге 8.6 представлен класс сервиса, который формирует список товаров и в котором определены два метода — для добавления нового товара и для получения списка товаров.

Листинг 8.6. Реализация двух сценариев использования приложения в классе `ProductService`

```
@Service
public class ProductService {

    private List<Product> products = new ArrayList<>();

    public void addProduct(Product p) {
        products.add(p);
    }
}
```

```

public List<Product> findAll() {
    return products;
}

}

```

ПРИМЕЧАНИЕ

Это упрощенная структура, позволяющая сосредоточиться на HTTP-методах. Как я уже говорил в главе 5, по умолчанию бины Spring имеют одиночную область определения, а веб-приложение создает несколько потоков (по одному для каждого запроса). В реальном приложении, когда сразу несколько клиентов будут добавлять новые товары, изменение списка, определенного как атрибут бина, приведет к состоянию гонки. Но пока что используем этот вариант: в следующих главах мы будем изменять список в базе данных, так что проблем не возникнет. Тем не менее вам следует помнить, что это порочная практика и не следует использовать подобный подход в реальных приложениях. При наличии потоков одиночные бины небезопасны!

Мы рассмотрим источники данных более подробно в главе 12, когда будем использовать базу данных — как это обычно делается в реальных приложениях. Но пока что сосредоточимся на обсуждаемой теме — HTTP-методах — и продолжим работу с нашими примерами.

Контроллер будет вызывать сценарии использования, реализованные в сервисе. Он получает от клиента данные о новом товаре и добавляет товар в список, вызывая сервис. Контроллер также получает перечень продуктов и передает его в представление. Ранее вы уже узнали, как реализовать эти функции. Прежде всего создадим класс `ProductController`, поместим его в пакет `controllers` и разрешим контроллеру внедрять бин сервиса. Определение контроллера представлено в листинге 8.7.

Листинг 8.7. Класс `ProductController`, применявший сервис для вызова сценариев использования

```

@Controller
public class ProductsController {

    private final ProductService productService;

    public ProductsController(
        ProductService productService) { ←
        this.productService = productService;
    }
}

```

Чтобы получить бин сервиса из контекста Spring, мы используем DI в конструкторе контроллера

Теперь мы реализуем первый сценарий использования: выведем список товаров на веб-страницу. Это простая функциональность. Для передачи данных из контроллера в представление мы воспользуемся параметром типа `Model`, как в разделе 8.1. Реализация этого действия представлена в листинге 8.8.

Листинг 8.8. Передача списка товаров в представление

```

@Controller
public class ProductsController {

    private final ProductService productService;           Связываем действие
                                                       контроллера с путем /products.

    public ProductsController(ProductService productService) {   Аннотация @RequestMapping
        this.productService = productService;                     по умолчанию подразумевает
    }                                                       использование HTTP-метода GET

    @RequestMapping("/products")
    public String viewProducts(Model model) {             Определяем параметр типа
                                                       Model для передачи данных
                                                       в представление

        var products = productService.findAll();           Получаем от сервиса список товаров
        model.addAttribute("products", products);          Передаем список товаров
                                                       в представление

        return "products.html";                         Возвращаем имя
                                                       представления — представление
                                                       будет получено и воспроизведено
                                                       диспетчером серверов
    }
}

```

Чтобы представление выводило на экран список товаров, создадим в папке проекта `resources/templates` файл `products.html`, как в разделе 8.1. Содержимое `products.html`, который получает переданный контроллером список товаров и выводит его в виде HTML-таблицы, показано в листинге 8.9.

Листинг 8.9. Представление списка товаров на веб-странице

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">           Определяем префикс th,
                                                               чтобы использовать
                                                               возможности Thymeleaf

    <head>
        <meta charset="UTF-8">
        <title>Home Page</title>
    </head>
    <body>
        <h1>Products</h1>

        <h2>View products</h2>                                         Используем функцию Thymeleaf th:each,
                                                               чтобы выполнить итерации по всей
                                                               коллекции товаров и вывести строку
                                                               таблицы для каждого товара в списке

        <table>
            <thead>
                <th>PRODUCT NAME</th>
                <th>PRODUCT PRICE</th>
            </thead>
            <tbody>
                <tr th:each="p: ${products}">           Даем таблице
                                                       статический заголовок
                    <td th:text="${p.name}"></td>
                    <td th:text="${p.price}"></td>          В каждой строке таблицы выводим
                                                       наименование и цену товара
                </tr>
            </tbody>
        </table>
    </body>
</html>

```

На рис. 8.10 показана диаграмма Spring MVC для процесса обработки пути /products с HTTP-методом GET:

1. Клиент отправляет HTTP-запрос для пути /products.
2. Диспетчер сервлетов с помощью карты обработчиков находит действие контроллера, которое нужно вызвать для пути /products.
3. Диспетчер вызывает это действие.
4. Контроллер запрашивает у сервиса список товаров и передает его для формирования представления.
5. Представление формируется в виде HTTP-ответа.
6. HTTP-ответ передается клиенту.

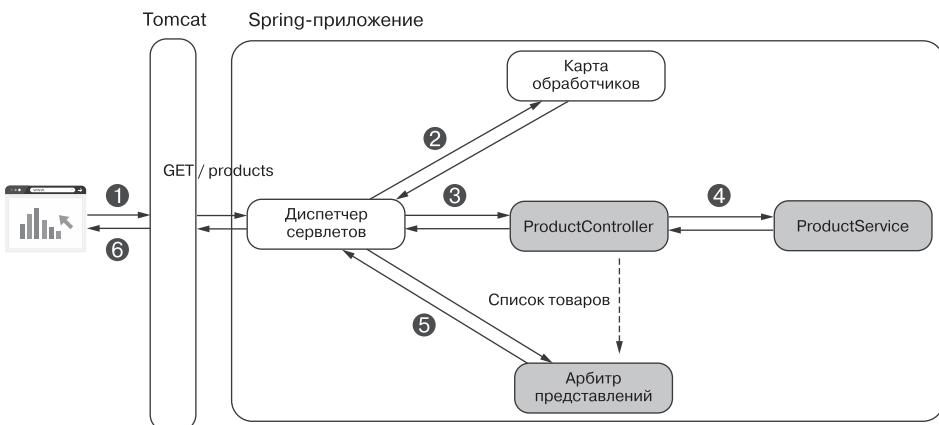


Рис. 8.10. При обращении по пути /products с HTTP-методом GET контроллер получает у сервиса список товаров и передает его в представление. HTTP-ответ формируется в виде HTML-таблицы с продуктами из этого списка

Но прежде, чем протестировать функционал приложения, мы все еще должны реализовать второй сценарий использования. Не имея возможности включить товар в список, мы увидим только пустую таблицу. Впишем в контроллер еще одно действие, позволяющее добавлять товары в список. Определение этого действия представлено в листинге 8.10.

Листинг 8.10. Метод действия для добавления товара

```
@Controller
public class ProductsController {
```

```
// остальной код
```

```
@RequestMapping(path = "/products",
method = RequestMethod.POST)
```

Связываем действие контроллера с путем /products. С помощью атрибута аннотации @RequestMapping изменяем HTTP-метод на POST

```
public String addProduct(  
    @RequestParam String name, | Из параметров запроса получаем наименование  
    @RequestParam double price, | и цену добавляемого товара  
    Model model  
) {  
    Product p = new Product();  
    p.setName(name);  
    p.setPrice(price);  
    productService.addProduct(p); | Создаем новый экземпляр Product и добавляем  
                                | его в список с помощью метода сервиса,  
                                | реализующего данный сценарий использования  
  
    var products = productService.findAll(); | Получаем список товаров  
    model.addAttribute("products", products); | и передаем его в представление  
  
    return "products.html"; <-- | Возвращаем имя представления,  
} | которое должно быть сформировано
```

Для определения HTTP-метода мы использовали атрибут аннотации `@RequestMapping`. Если не указать в ней метод, `@RequestMapping` по умолчанию использует GET. Но поскольку для любого HTTP-вызыва необходиимо знать и путь, и метод, лучше всегда прописывать и то и другое. Поэтому вместо `@RequestMapping` разработчики обычно используют специальные аннотации для каждого HTTP-метода. В приложениях для связи GET-запросов с действиями вам часто будут встречаться аннотации `@GetMapping`, для POST-запросов — `@PostMapping` и т. д. Мы тоже изменим наш пример, чтобы использовать специализированные аннотации. В листинге 8.11 представлен полный код контроллера, включая изменения аннотаций, связывающих запросы с действиями.

Листинг 8.11. Класс ProductController

```
@Controller  
public class ProductsController {  
  
    private final ProductService productService;  
  
    public ProductsController(ProductService productService) {  
        this.productService = productService;  
    }  
  
    @GetMapping("/products") ← Аннотация @GetMapping связывает действие контроллера с HTTP-запросом GET, имеющим заданный путь  
    public String viewProducts(Model model) {  
        var products = productService.findAll();  
        model.addAttribute("products", products);  
  
        return "products.html";  
    }  
  
    @PostMapping("/products") ← Аннотация @PostMapping связывает действие контроллера с HTTP-запросом POST, имеющим заданный путь
```

```

public String addProduct(
    @RequestParam String name,
    @RequestParam double price,
    Model model
) {
    Product p = new Product();
    p.setName(name);
    p.setPrice(price);
    productService.addProduct(p);

    var products = productService.findAll();
    model.addAttribute("products", products);

    return "products.html";
}
}

```

Мы также изменим представление, чтобы пользователь мог вызывать действие контроллера для HTTP-метода POST и добавлять товары в список. Для формирования этого запроса мы создадим HTML-форму. Новый код страницы показан в листинге 8.12, а результат его выполнения — на рис. 8.11.

Листинг 8.12. HTML-форма в конце списка товаров, позволяющая добавлять новые товары

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8">
        <title>Home Page</title>
    </head>
    <body>

        <!-- Остальной код -->

        <h2>Add a product</h2>
        <form action="/products" method="POST"> ← При передаче этой HTML-формы создается POST-запрос для пути /products
            Name: <input
                type="text"
                name="name"><br /> ← Данный компонент <input> позволяет пользователю ввести наименование товара. Значение этого компонента передается в виде параметра запроса с ключом name
            Price: <input
                type="number"
                step="any"
                name="price"><br /> ← Данный компонент <input> позволяет пользователю ввести цену товара. Значение этого компонента передается в виде параметра запроса с ключом price
            <button type="submit">Add product</button> ← Эта кнопка позволяет пользователю отправить форму
        </form>
    </body>
</html>

```

Теперь запустим приложение и проверим, как оно функционирует. При открытии страницы приложения по адресу <http://localhost:8080/products> у нас должна

появиться возможность добавлять новые товары и просматривать список уже включенных продуктов. Результат работы приложения показан на рис. 8.11.

Products

View products

PRODUCT NAME	PRODUCT PRICE
beer	5.0
chocolate	4.0

В этой HTML-таблице вы увидите все созданные товары

Add a product

Name: Price:

В этой HTML-форме можно добавить новый товар. Чтобы сформировать запрос на добавление товара, нужно заполнить поля ввода и нажать кнопку Add product

Рис. 8.11. Результат работы готового приложения. Пользователь видит на странице товары, представленные как HTML-таблица, и может добавлять новые с помощью HTML-формы

В примере я использовал уже знакомую вам из подраздела 8.1.2 аннотацию `@RequestParameter`, чтобы показать, как именно клиент передает данные. Но иногда Spring позволяет не писать этот код. Например, можно передать параметр типа `Product` непосредственно в действие контроллера, как показано в листинге 8.13. Поскольку имена параметров запроса совпадают с именами атрибутов класса `Product`, Spring знает, какие параметры каким атрибутам соответствуют, и автоматически создает объект. Для тех, кто уже хорошо знаком со Spring, это отличная новость, ведь тогда не придется писать лишние строки кода. Но начинающие могут запутаться во всех этих деталях. Предположим, вам попадется статья с примером, в котором используется подобный синтаксис. Вам может показаться непонятным, откуда там взялся экземпляр `Product`. Если вы, едва начав изучать Spring, окажетесь в подобной ситуации, советую вспомнить: синтаксис Spring построен таким образом, чтобы скрыть как можно больше кода. Если где-то вам встретится не вполне понятная конструкция, поищите разъяснения в спецификации фреймворка.

Я выделил это незначительное изменение в проект `sq-ch8-ex6`, чтобы вы могли его проверить и сравнить с проектом `sq-ch8-ex5`.

Листинг 8.13. Передача модели напрямую в действие контроллера в качестве параметра

```
@Controller
public class ProductsController {

    // остальной код

    @PostMapping("/products")
    public String addProduct(
        Product p,
        Model model
    ) {
        productService.addProduct(p); ← Класс модели можно передавать непосредственно
                                         в действие контроллера как параметр. Spring
                                         знает, что в этом случае нужно создать
                                         экземпляр, атрибуты которого соответствуют
                                         параметрам запроса. Для класса модели должен
                                         быть определен конструктор по умолчанию,
                                         чтобы Spring мог создать экземпляр этого класса
                                         перед вызовом метода действия

        var products = productService.findAll();
        model.addAttribute("products", products);

        return "products.html";
    }
}
```

РЕЗЮМЕ

- Страницы современных веб-приложений являются динамическими (их также называют динамическими представлениями). Содержимое динамической страницы меняется в зависимости от запроса.
- Чтобы знать, что отображать, динамическое представление получает изменяющиеся данные от контроллера.
- В Spring реализован простой способ создания динамических страниц для веб-приложений с помощью таких шаблонизаторов, как Thymeleaf. Кроме Thymeleaf, есть и другие варианты, например Mustache, FreeMarker и Java Server Pages (JSP).
- Шаблонизатор — это зависимость, благодаря которой в приложении появляется возможность легко получать данные, передаваемые контроллером, и выводить их в представлении.
- Клиент может отправлять данные на сервер в виде параметров запроса или переменных пути. Действие контроллера получает эти данные в виде параметров аннотации `@RequestParam` или `@PathVariable` соответственно.
- Параметры запроса не являются обязательными.
- Переменные пути следует использовать только для обязательных данных, передаваемых клиентом.
- HTTP-запрос характеризуется HTTP-методом и путем. HTTP-метод описывается ключевым словом, соответствующим намерениям клиента. На практике,

как правило, вам будут встречаться следующие HTTP-методы: GET, POST, PUT, PATCH и DELETE:

- метод GET соответствует намерению клиента получить данные, не изменяя их на стороне бэкенда;
 - метод POST характеризует намерение клиента добавить новые данные на стороне сервера;
 - метод PUT отражает намерение клиента полностью изменить конкретную запись данных на стороне бэкенда;
 - метод PATCH выражает намерение клиента частично изменить запись данных на стороне бэкенда;
 - метод DELETE соответствует намерению клиента удалить данные на стороне бэкенда.
- Для передачи данных непосредственно из HTML-формы в браузере можно применять только HTTP-методы GET и POST. Чтобы использовать остальные методы, такие как DELETE и PUT, нужно написать код вызова на клиентском языке программирования, таком как JavaScript.

Области веб-видимости бинов Spring

В этой главе

- ✓ Что такое области веб-видимости бинов Spring.
- ✓ Как реализовать простой функционал для веб-приложений.
- ✓ Как настроить перенаправление с одной страницы веб-приложения на другую.

В главе 5 мы рассмотрели области видимости бинов. Вы узнали, что в Spring жизненный цикл бина зависит от того, как этот бин был объявлен в контексте. Далее мы добавим еще несколько способов управления бинами в контексте Spring. Вы увидите, что у фреймворка есть свои способы контроля экземпляров в веб-приложениях и отправной точкой для них является HTTP-запрос. Ну правда же, Spring неимоверно крут? В любом Spring-приложении можно объявить бин, относящийся к одной из следующих областей видимости:

- *одиночный бин* — область видимости бинов в Spring по умолчанию. Фреймворк однозначно идентифицирует каждый такой экземпляр в контексте по его имени;
- *прототип* — область видимости бина в Spring, в случае которой фреймворк управляет только типом объектов и создает новый экземпляр класса для каждого запроса (сразу из контекста либо посредством монтажа или автомонтажа).

В этой главе вы узнаете, что в веб-приложениях можно использовать и другие области видимости, относящиеся только к веб-приложениям, — области веб-видимости бинов. Они бывают следующих типов:

- *область видимости в рамках запроса* — Spring создает отдельный экземпляр класса бина для каждого HTTP-запроса. Конкретный экземпляр существует только для конкретного HTTP-запроса;
- *область видимости в рамках сессии* — Spring создает экземпляр и хранит его в памяти сервера в течение всей HTTP-сессии. Фреймворк связывает этот экземпляр в контексте с сессией данного клиента;
- *область видимости в рамках приложения* — экземпляр является уникальным в контексте приложения и доступен все время работы приложения.

Чтобы вы поняли, как эти области работают в Spring-приложении, мы рассмотрим пример, где создадим функционал для аутентификации. В большинстве современных веб-приложений пользователю предоставляется возможность зарегистрироваться и войти в учетную запись, так что данный пример вполне актуален с практической точки зрения.

В разделе 9.1 мы создадим бин с областью видимости в рамках запроса, чтобы получить учетные данные для регистрации пользователя, и убедимся, что приложение использует их только для запроса на аутентификацию. Затем, в разделе 9.2, мы используем бин с областью видимости в рамках сессии, чтобы хранить в нем данные, которые нужно помнить о зарегистрированном пользователе, пока тот не вышел из приложения. В разделе 9.3 нам понадобится бин с областью видимости в рамках приложения, чтобы посчитать количество аутентификаций. На рис. 9.1 показаны все этапы создания этого продукта.

9.1. ИСПОЛЬЗОВАНИЕ БИНОВ С ОБЛАСТЬЮ ВИДИМОСТИ В РАМКАХ ЗАПРОСА В ВЕБ-ПРИЛОЖЕНИЯХ SPRING

Научимся использовать в веб-приложениях Spring бины с областью видимости в рамках запроса. Как вы уже знаете из глав 7 и 8, основная работа веб-приложений состоит в обработке HTTP-запросов и передаче ответов на них. Поэтому возможность создавать бины с жизненным циклом, привязанным к HTTP-запросу, значительно упрощает выполнение некоторых функций веб-приложений.

Бин с областью видимости в рамках запроса — это объект, управляемый Spring, для которого фреймворк создает новый экземпляр в рамках каждого HTTP-запроса. Приложение может использовать такой экземпляр только для того запроса, для которого он был создан. Для каждого следующего HTTP-запроса (от того же или другого клиента) создается и используется новый экземпляр аналогичного класса (рис. 9.2).

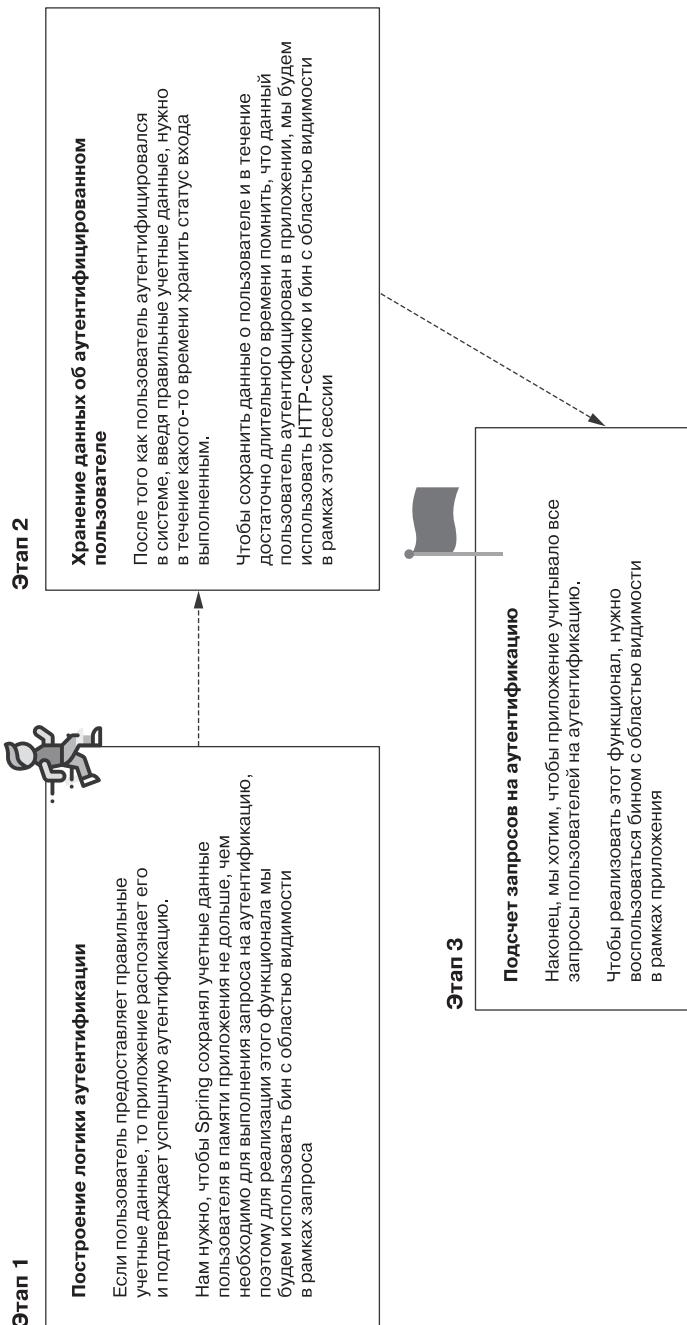


Рис. 9.1. Функционал аутентификации создается в три этапа. На каждом этапе разработки мы будем использовать бины с разной областью видимости. В разделе 9.1 для построения логики аутентификации мы воспользуемся бином с областью видимости в рамках запроса — это избавит нас от риска сохранения учетных данных после того, как запрос завершится. Затем мы решим, какие данные нужно сохранить для вошедшего пользователя в бине с областью видимости в рамках сессии. И в конце мы создадим функционал для подсчета всех запросов на аутентификацию. Мы будем хранить количество запросов в бине с областью видимости в рамках приложения

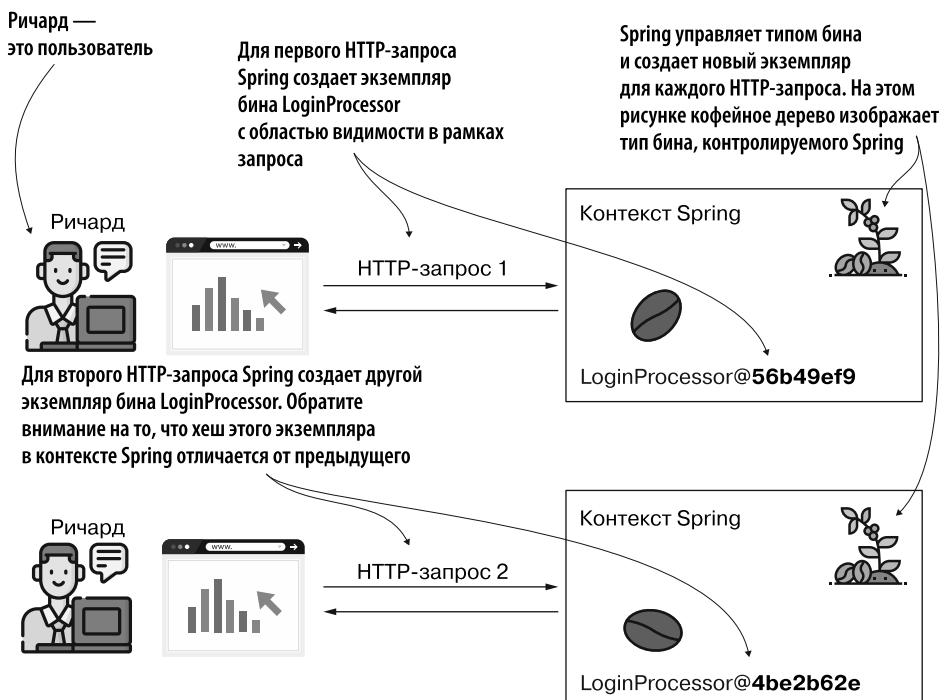


Рис. 9.2. Для каждого HTTP-запроса Spring создает новый экземпляр бина с областью видимости в рамках запроса. Используя такие бины, вы можете быть уверены, что данные, сохраняемые в нем, будут доступны только для того HTTP-запроса, для которого бин был создан. Spring управляет типом бина (кофейное дерево) и использует его, чтобы получать экземпляры (кофейные зерна) для каждого следующего запроса

Продемонстрируем использование бина с областью видимости в рамках запроса на примере. Мы создадим веб-приложение с функцией аутентификации и воспользуемся бином с областью видимости в рамках запроса для управления учетными данными пользователей, предоставляемыми для входа.

ГЛАВНЫЕ ОСОБЕННОСТИ БИНОВ С ОБЛАСТЬЮ ВИДИМОСТИ В РАМКАХ ЗАПРОСА

Прежде чем углубиться в создание Spring-приложения на основе бинов с областью видимости в рамках запроса, я бы хотел кратко перечислить главные особенности их использования. Знание их специфики поможет вам выбирать нужную область видимости при разработке реальных приложений.

Принимая такие решения, учитывайте наиболее важные аспекты — они перечислены в следующей таблице.

Факты	Следствия	Что учесть	Чего избегать
Spring создает новый экземпляр бина для каждого HTTP-запроса от каждого клиента	В процессе выполнения приложения Spring создает в его памяти множество экземпляров этого бина	Как правило, количество экземпляров не является большой проблемой, так как время их жизни невелико. Они нужны приложению не больше, чем выполняется HTTP-запрос. После завершения HTTP-запроса приложение уничтожает эти экземпляры и их подбирает сборщик мусора	Главное — проследить, чтобы в таких запросах не выполнялась затратная по времени логика, обычно необходимая Spring для создания экземпляров (такая как получение данных из базы данных или вызов функции по сети). Страйтесь не писать логику в конструкторах таких бинов или методах с аннотацией @PostConstruct
Экземпляр бина с областью видимости в рамках запроса доступен только одному запросу	Экземпляры бинов с областью видимости в рамках запроса не годятся для многопоточных задач, так как доступны только для одного потока (того, к которому принадлежит запрос)	В атрибутах такого экземпляра можно сохранять данные, используемые в запросе	Не используйте методы синхронизации для атрибутов таких бинов. Эти методы не сработают и лишь снизят производительность приложения

ПРИМЕЧАНИЕ

Примеры с аутентификацией наподобие того, что рассматривается в этой главе, отлично подходят для учебных целей. Но в реальных приложениях пострайтесь по возможности не писать собственные системы авторизации и аутентификации. Для всего, что связано с авторизацией и аутентификацией, там используют Spring Security. Проект Spring Security (который является частью экосистемы Spring) упрощает разрабатываемые продукты и гарантирует, что вы не создадите (по ошибке) уязвимости при построении логики обеспечения безопасности на уровне приложения. Советую также вам прочитать еще одну мою книгу, *Spring Security in Action* (Manning, 2020), в которой я подробно описываю использование Spring Security для защиты Spring-приложений.

Для простоты будем считать, что учетные данные всех пользователей встроены в приложение. На практике эта информация находится в базе данных; к тому же пароли хранятся в зашифрованном виде. Но мы пока что сосредоточимся только на теме этой главы — бинах Spring с областями веб-видимости. Далее, в главах 11 и 12, вы более подробно познакомитесь и с базами данных.

Создадим проект Spring Boot и внедрим необходимые зависимости. Этот пример находится в проекте `sq-ch9-ex1`. Зависимости можно добавить непосредственно при создании проекта (например, с помощью start.spring.io) или позднее, в файле `pom.xml`. В данном примере мы будем использовать веб-зависимость и Thymeleaf в качестве шаблонизатора (как в главе 8). Зависимости, которые нужно разместить в `pom.xml`, показаны в следующем фрагменте кода:

```
<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>
```

Мы создадим страницу с формой аутентификации, в которой нужно ввести имя пользователя и пароль. Приложение будет сравнивать их с набором имеющихся у него учетных данных (в моем случае это пользователь `natalie` с паролем `password`). Если предоставленные учетные данные правильные (соответствуют тем, которые есть в приложении), на странице под формой аутентификации появится сообщение `You are now logged in` (Вы вошли). Если же они неверны, то в сообщении будет сказано `Login failed` (Вход не выполнен).

Как и в главах 7–8, нам нужно создать страницу (соответствующую представлению) и класс контроллера. Контроллер будет передавать сообщение о результатах аутентификации, которое нужно вывести в представлении (рис. 9.3).

В листинге 9.1 представлена HTML-страница для аутентификации пользователя, которая является представлением приложения. Как вы уже знаете из главы 8, ее нужно сохранить в папке `resources/templates` проекта Spring Boot. Назовем эту страницу `login.html`. Для вывода сообщения по результатам выполнения логики нам нужно передавать параметр из контроллера в приложение. Как видно из следующего листинга, я назвал этот параметр `message` и использовал синтаксис `${message}`, чтобы под формой аутентификации появился нужный текст.

Действию контроллера должен поступить HTTP-запрос (от диспетчера сервлетов, как известно нам из глав 7–8), так что создадим контроллер и действие, которое получает запрос от страницы, созданной в листинге 9.1. Определение класса контроллера представлено в листинге 9.2. Мы связали действие контроллера с корневым путем приложения (`/`). Я дал контроллеру имя `LoginController`.

Клиент передает HTTP-запрос с учетными данными пользователя

HTTP-запрос POST
/?username=natalie&password=password

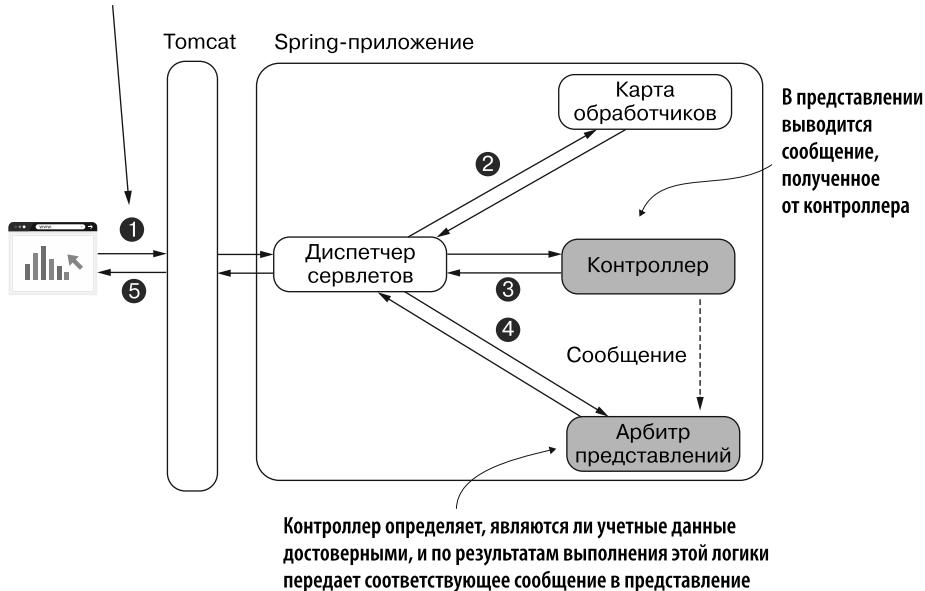


Рис. 9.3. Нам нужно создать контроллер и представление. В контроллере мы сформулируем действие, которое будет определять, являются ли учетные данные, переданные в запросе аутентификации, достоверными. Контроллер передает сообщение в представление, а представление выводит это сообщение на экран

Листинг 9.1. Код страницы аутентификации login.html

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org"> ← Определяем префикс th, чтобы использовать возможности шаблонизатора Thymeleaf
<head>
    <meta charset="UTF-8">
    <title>Login</title>
</head>
<body>
    <form action="/" method="post"> ← Создаем HTML-форму для отправки учетных данных пользователя на сервер
        Username: <input type="text" name="username" /><br />
        Password: <input type="password" name="password" /><br />
        <button type="submit">Log in</button> ← Поля ввода учетных данных — имени пользователя и пароля
    </form>
    <p th:text="${message}"></p> ← Когда пользователь щелкнет на кнопке Submit, клиент создаст HTTP-запрос типа POST с учетными данными этого пользователя
</body>
</html>
```

Под HTML-формой выводим сообщение с результатом запроса аутентификации

Листинг 9.2. Действие контроллера, связанное с корневым путем приложения

```
@Controller
public class LoginController {
    @GetMapping("/")
    public String loginGet() {
        return "login.html";
    }
}
```

Используем стереотипную аннотацию `@Controller`, чтобы показать, что данный класс является контроллером

Связываем действие контроллера с корневым путем приложения `/`

Возвращаем имя представления, которое должно быть сформировано приложением

Теперь у нас есть страница входа — можно писать логику аутентификации. Когда пользователь нажимает кнопку `Submit`, страница должна размещать под формой соответствующее сообщение: если введенные учетные данные правильные, то `You are now logged in`, если нет — то `Login failed` (рис. 9.4).

Приложение выводит форму, в которую пользователь вносит свои учетные данные для аутентификации

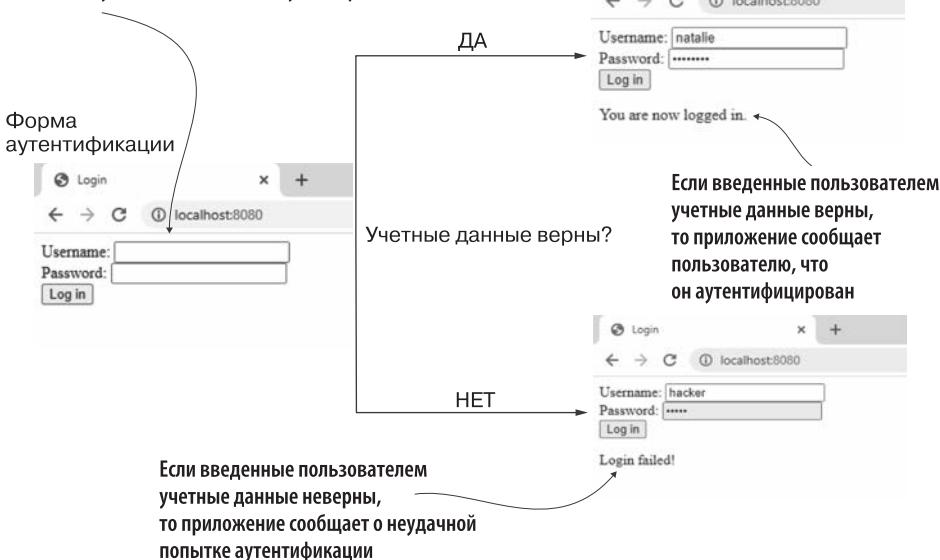


Рис. 9.4. Функциональность, реализованная в этом разделе. На странице выводится форма аутентификации пользователя. Если пользователь указывает правильные учетные данные, то выводится сообщение об успешном входе. Если данные неверны, то приложение говорит пользователю, что аутентификация не удалась

Для обработки HTTP-запроса типа POST, создаваемого HTML-формой, когда пользователь нажимает `Submit`, нужно добавить в `LoginController` еще одно действие. Данное действие принимает параметры запроса клиента (имя пользователя и пароль) и передает в представление сообщение о результатах аутентификации. В листинге 9.3 представлено определение действия контроллера. Мы свяжем его с HTTP-запросом аутентификации типа POST.

Обратите внимание, что мы не создавали логику аутентификации. В следующем листинге мы принимаем запрос и передаем сообщение в зависимости от содержимого переменной, представляющей результат запроса. Но значение этой переменной (в листинге 9.3 это `loggedIn`) всегда равно `false`. В последующих листингах раздела мы дополним данное действие, добавив туда логику аутентификации. Результат ее работы будет зависеть от учетных данных, переданных в запросе от клиента.

Листинг 9.3. Действие контроллера, выполняющее аутентификацию

```
@Controller
public class LoginController {

    @GetMapping("/")
    public String loginGet() {
        return "login.html";
    }

    @PostMapping("/")
    public String loginPost(
        @RequestParam String username,
        @RequestParam String password,
        Model model
    ) {
        boolean loggedIn = false; // Позже, когда мы напишем логику
        // аутентификации, в этой переменной будет
        // храниться результат обработки запроса на вход

        if (loggedIn) {
            model.addAttribute("message", "You are now logged in.");
        } else {
            model.addAttribute("message", "Login failed!");
        }

        return "login.html"; // Возвращаем имя представления. Это по-прежнему login.html,
        // так что мы остаемся на той же странице
    }
}
```

The diagram uses callout boxes to explain the annotations:

- `@GetMapping("/")`: Связываем действие контроллера с HTTP-запросом типа POST, отправляемым со страницы аутентификации
- `@PostMapping("/")`: Извлекаем учетные данные из параметров HTTP-запроса
- `@RequestParam`: Объявляем параметр типа Model, чтобы передавать в представление текст сообщения
- `boolean loggedIn = false;`: Позже, когда мы напишем логику аутентификации, в этой переменной будет храниться результат обработки запроса на вход
- `return "login.html";`: В зависимости от результата аутентификации, отправляем в представление то или иное сообщение

На рис. 9.5 наглядно представлена связь между созданным нами классом контроллера и представлением.

Теперь у нас есть контроллер и представление, но где здесь область видимости запроса? Единственный написанный нами класс — `LoginController`, и это одиночный бин, что относится к области видимости Spring по умолчанию. Нам не нужно изменять область видимости `LoginController`, так как в его атрибутах не хранятся никакие данные. Но, напомню, необходимо еще написать логику аутентификации. Логика аутентификации зависит от учетных данных пользователя, и относительно этих данных необходимо принять во внимание два момента.

1. Учетные данные пользователя — конфиденциальная информация. Ее нельзя хранить в памяти приложения дольше, чем обрабатывается запрос на аутентификацию.
2. Возможна ситуация, когда сразу несколько пользователей захотят аутентифицироваться, каждый со своими учетными данными.

Класс LoginController

```

@Controller
public class LoginController {
    @GetMapping("/")
    public String loginGet() {
        return "login.html";
    }

    @PostMapping("/")
    public String loginPost(@RequestParam String username,
                           @RequestParam String password,
                           Model model) {
        boolean loggedIn = true;
        if (loggedIn) {
            model.addAttribute("message", "You are now logged in.");
        } else {
            model.addAttribute("message", "Login failed!");
        }
        return "login.html";
    }
}

```

Страница login.html

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Login</title>
</head>
<body>
    <form action="/" method="post">
        Username: <input type="text" name="username" /><br />
        Password: <input type="password" name="password" /><br />
        <button type="submit">Log in</button>
    </form>
    <p th:text="${message}"></p>
</body>
</html>

```

Это действие контроллера вызывается при отправке формы

Действие контроллера
принимает учетные данные из параметров запроса
model.addAttribute("message", "You are now logged in.");
model.addAttribute("message", "Login failed!");

Действие контроллера передает в представление сообщение, содержимое которого зависит от результатов обработки запроса на аутентификацию.
Представление выводит сообщение под формой аутентификации

Рис. 9.5. Когда кто-нибудь отправляет на сервер форму аутентификации, диспетчер сервлетов вызывает соответствующее действие контроллера. Это действие извлекает учетные данные из параметров HTTP-запроса. В зависимости от результатов аутентификации контроллер передает в представление то или иное сообщение, а представление выводит это сообщение под HTML-формой

Учитывая эти особенности, нам нужно гарантировать, что при использовании бина для построения логики аутентификации у каждого HTTP-запроса будет свой уникальный экземпляр. Нам нужен бин с областью видимости в рамках запроса. Мы дополним приложение, как показано на рис. 9.5, добавив бин `LoginProcessor` с областью видимости в рамках запроса. Этот бин извлекает учетные данные из запроса и проверяет их (рис. 9.6).

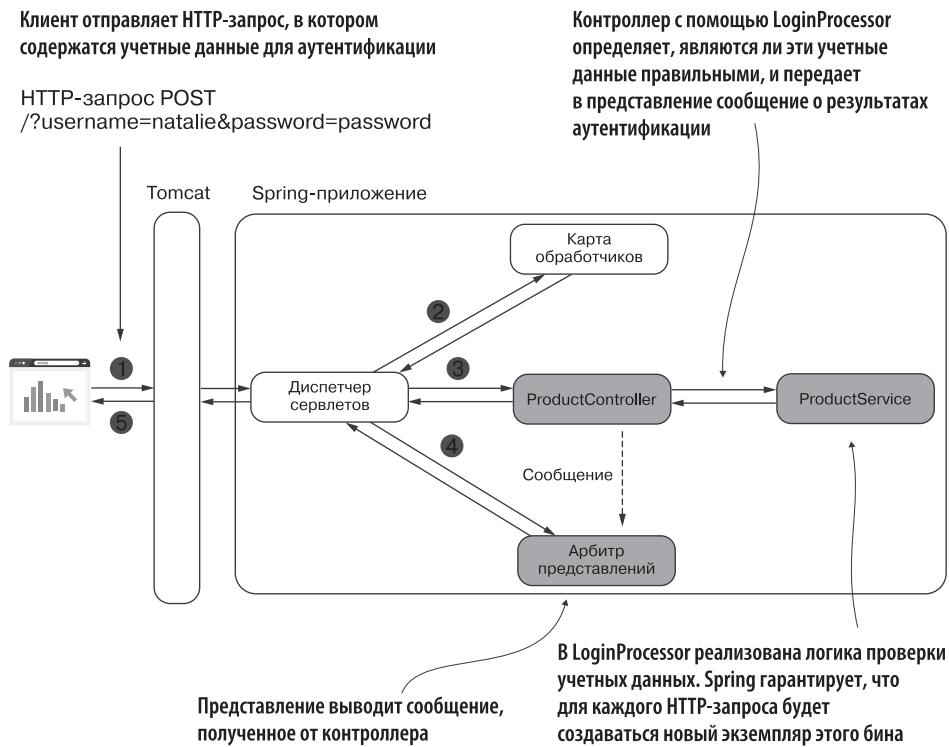


Рис. 9.6. Бин `LoginProcessor` имеет область видимости в рамках запроса. Spring гарантирует, что для каждого HTTP-запроса будет создаваться новый экземпляр этого бина. В `LoginProcessor` реализована логика аутентификации. Контроллер вызывает метод, выполняющий эту логику. Метод возвращает `true`, если учетные данные верны, и `false`, если нет. Сообщение, которое `LoginController` передает в представление, определяется тем, какое значение возвращает `LoginProcessor`

Реализация класса `LoginProcessor` представлена в листинге 9.4. Чтобы изменить область видимости бина, мы воспользовались аннотацией `@RequestScoped`. Разумеется, нам все равно нужно создать бин этого класса в контексте Spring с помощью аннотации `@Bean` в классе конфигурации либо с помощью стереотипной аннотации. Я решил снабдить класс стереотипной аннотацией `@Component`.

Листинг 9.4. Бин LoginProcessor с областью видимости в рамках запроса, реализующий логику аутентификации

```

@Component ←
@RequestScope ←
public class LoginProcessor {

    private String username;
    private String password; ← С помощью стереотипной аннотации
                                сообщает Spring, что это бин

    public boolean login() { ← С помощью аннотации @RequestScope
        String username = this.getUsername(); ограничиваем область видимости бина текущим
        String password = this.getPassword(); запросом. Теперь Spring будет создавать новый
                                            экземпляр класса для каждого HTTP-запроса

        if ("natalie".equals(username) && "password".equals(password)) { ← Учетные данные хранятся в бине в качестве атрибутов
            return true;
        } else {
            return false;
        }
    }

    // геттеры и сеттеры
}

```

В бине определен метод, в котором реализована логика аутентификации

Теперь можно запустить приложение, введя в адресной строке браузера адрес `localhost:8080`. На рис. 9.7 показано поведение приложения после того, как пользователь откроет эту страницу и введет корректные и некорректные учетные данные.

Форма аутентификации
Открыв веб-страницу, вы сначала увидите пустую форму для аутентификации

Учетные данные верны
Если ввести правильные учетные данные — имя пользователя `natalie` и пароль `password` — и нажать `Log in`, то приложение выведет сообщение об успешной аутентификации

Учетные данные неверны
При использовании неверных учетных данных приложение выводит сообщение `Login failed!` (Аутентификация не удалась!)

Рис. 9.7. Когда страница открывается в браузере, приложение выводит форму аутентификации. Если ввести правильные учетные данные, приложение выдаст сообщение об успешной аутентификации. Если учетные данные будут неверными, появится текст `Login failed!` (Аутентификация не удалась!)

9.2. ИСПОЛЬЗОВАНИЕ ОБЛАСТИ ВИДИМОСТИ В РАМКАХ СЕССИИ В ВЕБ-ПРИЛОЖЕНИЯХ SPRING

Рассмотрим бины с областью видимости в рамках сессии. Открыв веб-приложение и войдя в учетную запись, пользователь ожидает, что, пока он будет просматривать страницы приложения, оно будет помнить, что данный пользователь аутентифицирован. Бин с областью видимости в рамках сессии — это управляемый Spring объект, для которого фреймворк создает экземпляр, привязанный к текущей HTTP-сессии. Когда клиент посыпает запрос на сервер, сервер выделяет в памяти место для этого запроса на все время сессии, к которой данный запрос относится. Spring создает экземпляр бина с областью видимости в рамках сессии в начале HTTP-сессии для данного клиента. Этот экземпляр может многократно использоваться одним и тем же клиентом, пока HTTP-сессия остается активной. Данные в атрибутах бина с областью видимости в рамках сессии доступны для всех запросов клиента в рамках сессии. Такой способ хранения данных позволяет не потерять информацию о том, что делает пользователь, пока переходит по страницам приложения.

При всех запросах, передаваемых в рамках одной HTTP-сессии, клиент получает доступ к одному и тому же экземпляру бина. У каждого пользователя есть своя сессия, поэтому они получают доступ к собственным бинам, имеющим область видимости в рамках сессии.

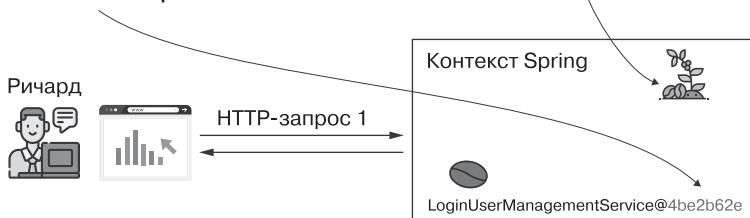
Потратим некоторое время на анализ рис. 9.8, на котором представлен бин с областью видимости в рамках сессии, и рис. 9.2, где показан бин с областью видимости в рамках запроса. На рис. 9.9 показан итог сравнения этих двух вариантов. Если в случае бина с областью видимости в рамках запроса Spring создает новый экземпляр для каждого HTTP-запроса, то во втором случае фреймворк выделяет единственный экземпляр для всей HTTP-сессии. Бины с областью видимости в рамках сессии позволяют хранить данные, используемые несколькими запросами одного и того же клиента.

Бины с областью видимости в рамках сессии позволяют реализовать, в частности, такие функции, как:

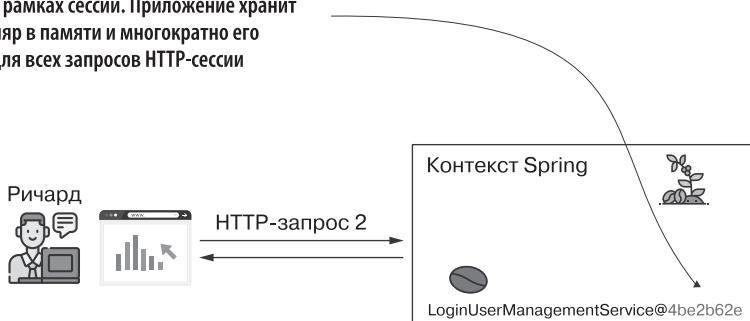
- **аутентификация.** В бине сохраняются данные об аутентифицированном пользователе в течение всего времени, пока он посещает различные страницы приложения и отправляет запросы;
- **корзина интернет-магазина.** Пользователь посещает разные страницы приложения в поисках товаров, которые он хочет добавить в корзину. Корзина запоминает все наименования, которые в нее поместил клиент.

Когда Ричард отправляет первый запрос, начинается новая HTTP-сессия и Spring создает новый экземпляр бина с областью видимости в рамках сессии. Приложение будет использовать его на протяжении всего этого времени

Бин с областью видимости в рамках сессии отмечен значком кофейного дерева. Spring отслеживает тип объекта, но создает и обслуживает несколько экземпляров этого объекта. Spring создает по одному экземпляру для каждой HTTP-сессии



Затем Ричард отправляет запрос на использование уже созданного экземпляра бина с областью видимости в рамках сессии. Приложение хранит этот экземпляр в памяти и многократно его использует для всех запросов HTTP-сессии



Когда Даниэла начинает свою HTTP-сессию, Spring создает экземпляр бина и для нее. В это время в контексте Spring также хранятся экземпляры этого объекта для других пользователей. Бин сессии Ричарда тоже здесь, и приложение продолжает использовать его для запросов клиента

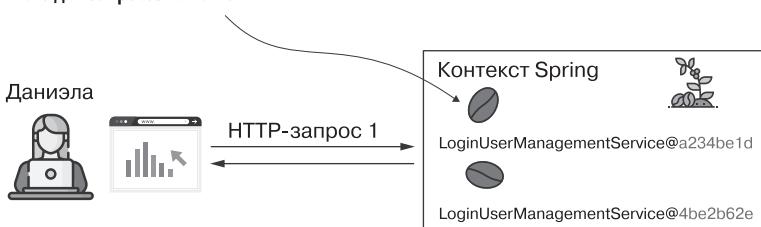
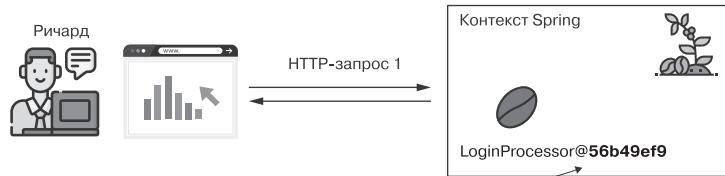
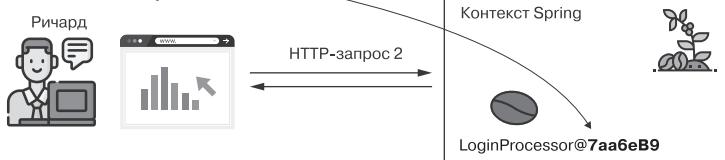
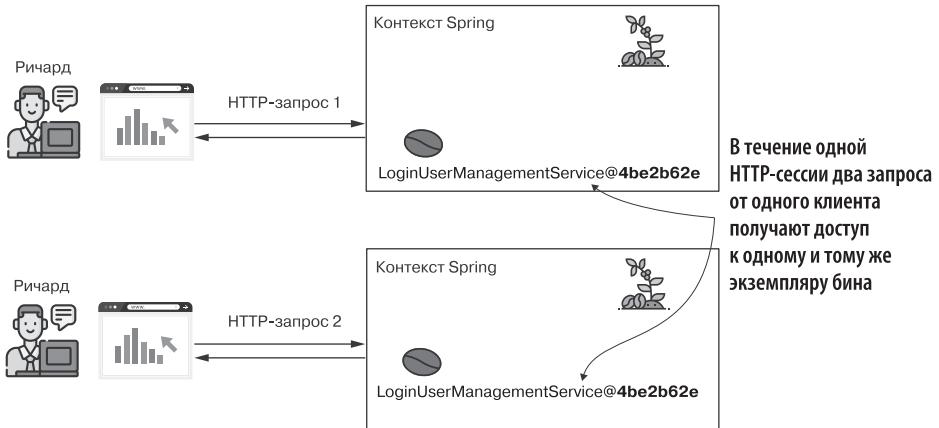


Рис. 9.8. Бин с областью видимости в рамках сессии можно хранить в контексте в течение всей HTTP-сессии клиентом. Spring создает экземпляр такого бина для каждой сессии, открываемой клиентом

Бины с областью видимости в рамках запроса

Для каждого запроса Spring создает отдельный экземпляр бина

**Бины с областью видимости в рамках сессии**

Каждый клиент создает свою HTTP-сессию и получает доступ к своему бину

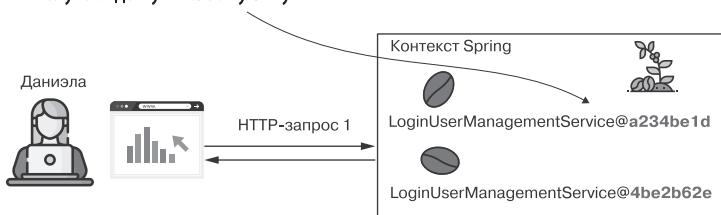


Рис. 9.9. Сравнение бинов с областью видимости в рамках запроса и в рамках сессии поможет вам наглядно представить различия между этими двумя областями видимости. Первые бины применяют, когда нужно создать новый экземпляр бина для каждого запроса. Вторые используются в тех случаях, когда бин (и все данные, которые в нем хранятся) должен быть доступен в течение всей HTTP-сессии данного клиента

ОСНОВНЫЕ СВОЙСТВА БИНОВ С ОБЛАСТЬЮ ВИДИМОСТИ В РАМКАХ СЕССИИ

Как и в случае бинов с областью видимости в рамках запроса, проанализируем главные характеристики бинов с областью видимости в рамках сессии — те, которые необходимо учитывать при использовании таких бинов в различных приложениях.

Факты	Следствия	Что учесть	Чего избегать
Экземпляры бинов с областью видимости в рамках сессии сохраняются в течение всей HTTP-сессии	Время жизни таких бинов больше, чем у бинов с областью видимости в рамках запроса, и они не так часто попадают в сборку мусора	Данные, сохраненные в бинах с областью видимости в рамках сессии, приложение помнит дольше	Не стоит хранить в сессии слишком много данных — это может привести к проблемам с производительностью. И тем более не следует помещать в атрибуты бинов с областью видимости в рамках сессии конфиденциальную информацию, такую как пароли, частные ключи и др.
Один экземпляр бина с областью видимости в рамках сессии может быть доступен нескольким запросам	Если один и тот же клиент сделает несколько конкурентных запросов, изменяющих данные в таком экземпляре, возможны проблемы многопоточности, например постоянные гонки	Возможно, чтобы избежать конкуренции, стоит воспользоваться механизмами синхронизации. Но я обычно рекомендую подумать, можно ли не допускать появления такой проблемы и оставить синхронизацию на самый крайний случай	
Бины с областью видимости в рамках сессии — это способ сделать данные доступными для нескольких запросов, сохраняя эти данные на стороне сервера	Реализуемая вами логика может потребовать запросов, зависящих друг от друга	Когда данные о состоянии хранятся в памяти приложения, клиенты становятся зависимыми от этого конкретного объекта приложения. Принимая решение о реализации какого-либо функционала посредством бина с областью видимости в рамках сессии, рассмотрите другие варианты хранения данных, которые вы хотите сделать общедоступными, например, не в сессии, а в базе данных, чтобы HTTP-запросы остались независимыми друг от друга	

Продолжим создавать бин с областью видимости в рамках сессии, чтобы приложение помнило о вошедшем пользователе и признавало его аутентифицированным на разных страницах. Таким образом, в этом примере вы изучите все важные моменты, которые следует знать при работе с реальными приложениями.

Внесем изменения в код приложения, созданного в разделе 9.1, и добавим в него страницу, которая открывается только для аутентифицированных пользователей. После того как пользователь аутентифицируется, приложение перенаправляет его на эту страницу. На ней будет выводиться приветствие с указанием имени пользователя и предложение выйти из приложения, перейдя по ссылке.

Для реализации этих изменений нужно выполнить следующие действия (рис. 9.10).

1. Создать бин с областью видимости в рамках сессии, чтобы хранить в нем данные об аутентифицированном пользователе.
2. Создать страницу, на которую пользователь может попасть только после аутентификации.
3. Проследить, чтобы пользователь не мог открыть страницу, созданную в пункте 2, предварительно не аутентифицировавшись.
4. После успешной аутентификации перенаправлять пользователя со страницы с формой на главную страницу.

Я вынес изменения этого примера в отдельный проект sq-ch9-ex2.

К счастью, в Spring, чтобы создать бин с областью видимости в рамках сессии, достаточно добавить к классу бина аннотацию `@SessionScope`. Создадим новый класс `LoggedUserManagementService` с областью видимости в рамках сессии, как показано в следующем листинге.

Листинг 9.5. Определение бина с областью видимости в рамках сессии для хранения данных об аутентифицированном пользователе

```

@Service ←
@SessionScope ←
public class LoggedUserManagementService {
    private String username;
    // геттеры и сеттеры
}

```

Добавляем стереотипную аннотацию `@Service`, чтобы Spring создал бин этого класса и добавил его в контекст

С помощью аннотации `@SessionScope` меняем область видимости бина на видимость в рамках сессии

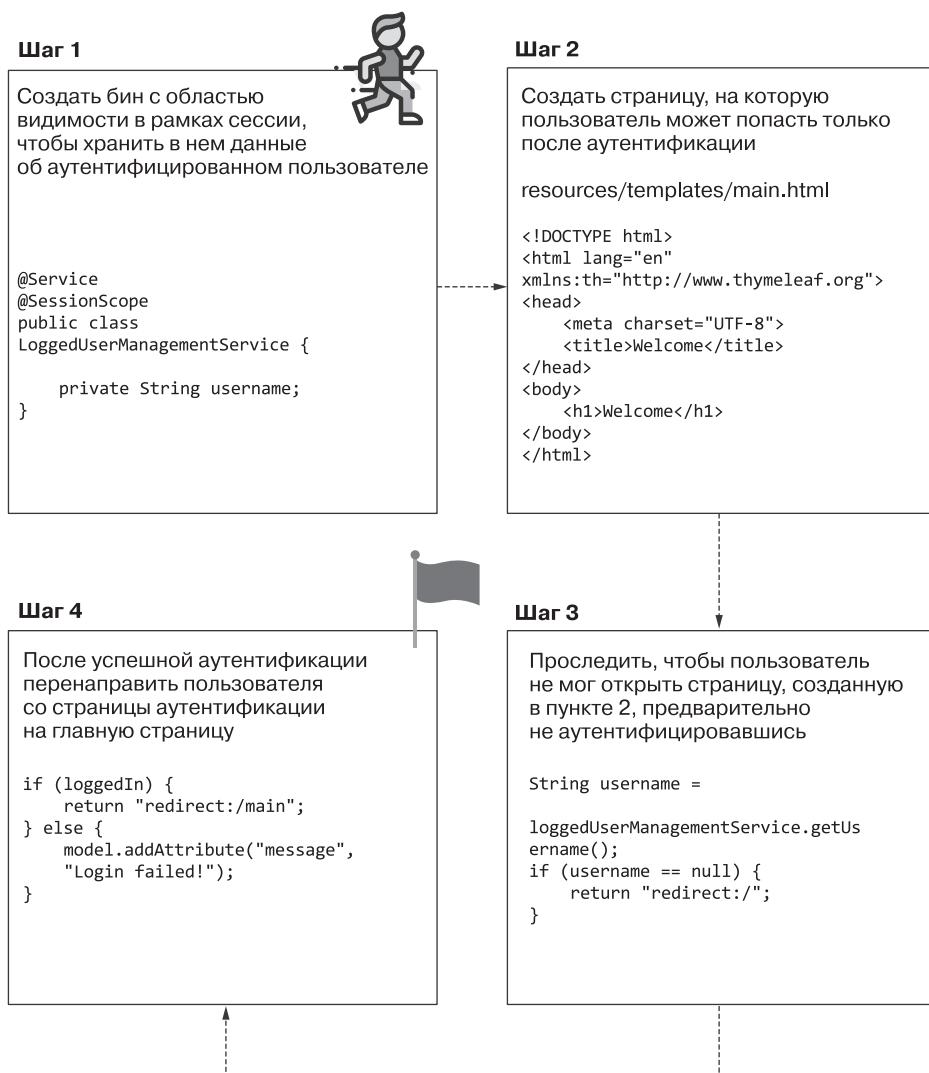


Рис. 9.10. С помощью бина, имеющего область видимости в рамках сессии, мы создадим раздел приложения, доступный только аутентифицированным пользователям. Приложение будет перенаправлять пользователя на эту страницу только после успешной аутентификации. Если он попытается открыть страницу не зарегистрировавшись, приложение вернет его в форму аутентификации

После каждой успешной аутентификации имя пользователя сохраняется в атрибуте бина `username`. Бин `LoggedUserManagementService` автомонтируется

к классу `LoginProcessor`, созданному в разделе 9.1 и отвечающему за логику аутентификации, как показано в следующем листинге.

Листинг 9.6. Применение бина `LoggedUserManagementService` в логике аутентификации

```

@Component
@RequestScope
public class LoginProcessor {

    private final LoggedUserManagementService loggedUserManagementService;

    private String username;
    private String password;

    public LoginProcessor( ← Автомонтируем бин LoggedUserManagementService
        LoggedUserManagementService loggedUserManagementService) {
        this.loggedUserManagementService = loggedUserManagementService;
    }

    public boolean login() {
        String username = this.getUsername();
        String password = this.getPassword();

        boolean loginResult = false;
        if ("natalie".equals(username) && "password".equals(password)) {
            loginResult = true;
            loggedUserManagementService.setUsername(username); ←
        }
        return loginResult;
    }

    // геттеры и сеттеры
}

```

Сохраняем имя пользователя в бине
LoggedUserManagementService

Обратите внимание: бин `LoginProcessor` по-прежнему имеет область видимости в рамках запроса. Spring продолжает создавать для нового запроса свой экземпляр этого бина. Просто теперь для выполнения логики аутентификации каждый раз нужно получать значения атрибутов `username` и `password`.

Поскольку бин `LoggedUserManagementService` имеет область видимости в рамках сессии, значение его атрибута `username` доступно на протяжении всей HTTP-сессии. Мы можем использовать это значение, чтобы определить, есть ли на сайте сейчас аутентифицированные пользователи и кто именно. Нам не приходится беспокоиться о том, что пользователей может быть несколько: фреймворк приложения обеспечит соединение каждого

HTTP-запроса с соответствующей сессией. Процедура аутентификации наглядно показана на рис. 9.11.

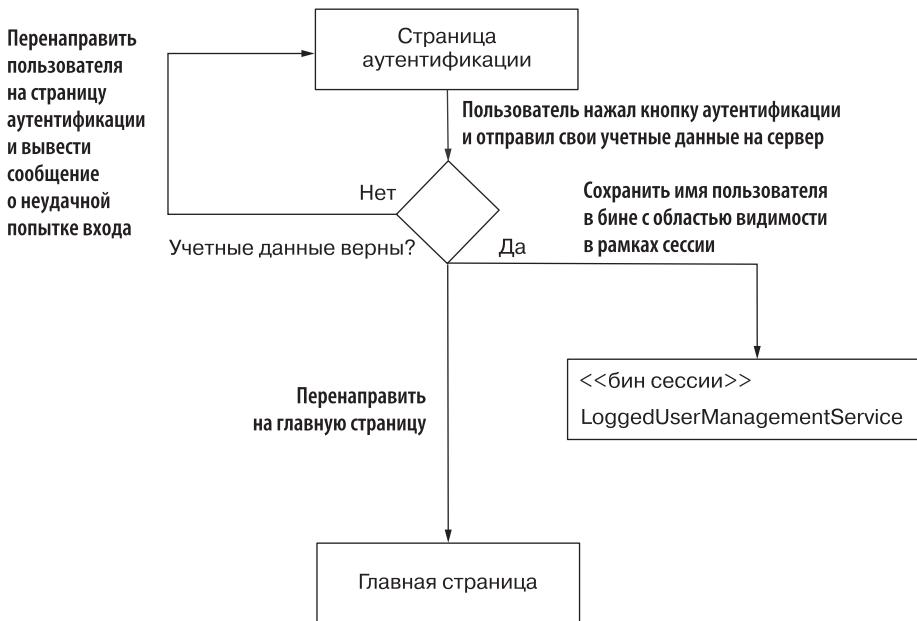


Рис. 9.11. Процедура аутентификации, реализованная в данном примере.

После того как пользователь передаст на сервер свои учетные данные, начинается процесс аутентификации. Если данные верны, имя пользователя сохраняется в бине с областью видимости в рамках сессии, и приложение перенаправляет пользователя на главную страницу. Если же данные неверны, приложение возвращает пользователя на страницу аутентификации и выводит сообщение о неудачной попытке входа

Теперь создадим новую страницу и проследим, чтобы она была доступна только для аутентифицированных пользователей. Для этой страницы мы разрабатываем новый контроллер (который назовем `MainController`), где определим действие и свяжем его с путем `/main`. Чтобы пользователь получал доступ к странице, расположенной по этому пути, только после успешной аутентификации, мы будем проверять, хранится ли в `LoggedUserManagementService` имя пользователя, и, если нет, будем перенаправлять пользователя на страницу аутентификации. Для реализации этого функционала действие контроллера должно возвращать строку `redirect:` и путь, по которому это действие перенаправляет пользователя. Логика вывода главной страницы наглядно представлена на рис. 9.12.



Рис. 9.12. Пользователь может получить доступ к главной странице только после аутентификации. Когда приложение аутентифицирует пользователя, оно сохраняет его имя в бине сессии. Таким образом, приложение знает, что данный пользователь аутентифицирован. Когда кто-нибудь пытается получить доступ к главной странице, но в бине его сессии имени пользователя не будет (ведь он не аутентифицирован), приложение перенаправляет его на страницу с формой входа

Класс `MainController` представлен в листинге 9.7.

Листинг 9.7. Класс `MainController`

```

@Controller
public class MainController {
    private final LoggedUserManagementService loggedUserManagementService;
    Автомонтируем бин LoggedUserManagementService,
    чтобы можно было узнать, аутентифицирован ли пользователь

    public MainController( ←
        LoggedUserManagementService loggedUserManagementService) {
        this.loggedUserManagementService = loggedUserManagementService;
    }
}

```

```

@GetMapping("/main")
public String home() {
    String username = ← Получаем значение username — если
        loggedInUserManagementService.getUsername();   пользователь аутентифицирован, то оно
                                                        не должно быть равно null

    if (username == null) { ← Если пользователь не аутентифицирован,
        return "redirect:/";   перенаправляем его на страницу аутентификации
    }

    return "main.html"; ← Если пользователь аутентифицирован,
}                                возвращаем представление главной страницы
}

```

Теперь нужно создать в папке `resources/templates` проекта Spring Boot файл `main.html`, определяющий представление главной страницы. Ее содержимое показано в листинге 9.8.

Листинг 9.8. Содержимое страницы main.html

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Welcome</title>
</head>
<body>
    <h1>Welcome</h1>
</body>
</html>

```

Дать пользователю возможность выйти из приложения также легко. Для этого нужно просто присвоить атрибуту `username` бина сессии `LoggedInUserManagementService` значение `null`. Создадим на странице ссылку для выхода из приложения, а также добавим имя аутентифицированного пользователя в сообщение приветствия. Соответствующие изменения в файле `main.html`, описывающем представление, показаны в листинге 9.9.

Листинг 9.9. Добавление ссылки для выхода из приложения на страницу main.html

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>Login</title>
</head>
<body>
    <h1>Welcome, <span th:text="${username}"></span></h1> ← Получаем от контроллера
    <a href="/main?Logout">Log out</a> ← и выводим его на странице
                                                в приветствии
</body>
</html>

```

Добавляем на страницу ссылку, которая отправляет HTTP-запрос с параметром `Logout`. Получив этот параметр, контроллер удалит из сессии значение атрибута `username`

Чтобы нужный нам функционал заработал, потребуется кое-что изменить не только на странице `main.html`, но и в контроллере. В листинге 9.10 показано, как действие контроллера получает параметр запроса о выходе из приложения и передает в представление имя пользователя, где оно будет выведено на веб-странице.

Листинг 9.10. Выход пользователя из приложения с помощью параметра запроса `Logout`

```
@Controller
public class MainController {

    // Остальной код

    @GetMapping("/main")
    public String home(
        @RequestParam(required = false) String Logout, ← Извлекаем из запроса
        Model model ← Добавляем параметр Model, чтобы передать
    ) {                                         имя пользователя в представление
        if (Logout != null) { ← Если в запросе есть параметр
            loggedInUserManagementService.setUsername(null); Logout, удаляем из бина
        }                                         LoggedUserManagementService
                                                имя пользователя
        String username = loggedInUserManagementService.getUsername();

        if (username == null) {
            return "redirect:/";
        }

        model.addAttribute("username" , username); ← Передаем имя пользователя
        return "main.html";
    }
}
```

И в завершение работы над приложением внесем корректизы в `LoginController`, чтобы перенаправлять пользователей на главную страницу после аутентификации. Для этого нам нужно изменить действие `LoginController`, как показано в листинге 9.11.

Листинг 9.11. Перенаправление пользователя на главную страницу после аутентификации

```
@Controller
public class LoginController {

    // Остальной код

    @PostMapping("/")
    public String loginPost(
        @RequestParam String username,
        @RequestParam String password,
        Model model
```

```

) {
    loginProcessor.setUsername(username);
    loginProcessor.setPassword(password);
    boolean loggedIn = loginProcessor.login();

    if (loggedIn) { ←
        return "redirect:/main";
    }

    model.addAttribute("message", "Login failed!");
    return "login.html";
}
}

```

После успешной аутентификации
приложение перенаправляет
пользователя на главную страницу

Теперь можно запустить приложение и проверить, как работает аутентификация. Если ввести правильные учетные данные, приложение перенаправит вас на главную страницу (рис. 9.13). Если щелкнуть на ссылке **Log out**, приложение вернет на страницу аутентификации. Если попытаться открыть главную страницу без аутентификации, приложение отправит вас на страницу с формой входа.

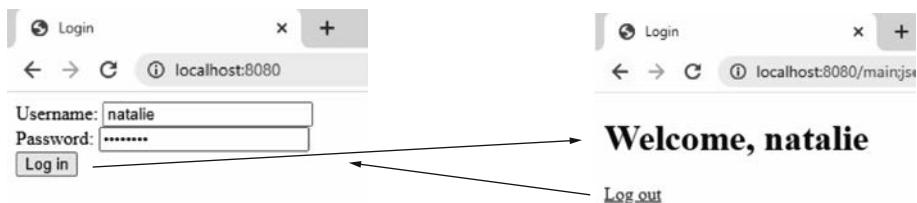


Рис. 9.13. Перенаправление пользователя между двумя страницами: после того как пользователь аутентифицировался, приложение отправляет его на главную страницу. Пользователь может щелкнуть на ссылке выхода из приложения, и тогда он будет перенаправлен обратно к форме аутентификации

9.3. ИСПОЛЬЗОВАНИЕ ОБЛАСТИ ВИДИМОСТИ В РАМКАХ ВСЕГО ВЕБ-ПРИЛОЖЕНИЯ SPRING

В этом разделе мы рассмотрим область видимости в рамках приложения. Я хочу лишь отметить, что она существует, рассказать, как она работает, и особо подчеркнуть, что лучше не использовать ее в реальных приложениях.

Бин с областью видимости в рамках приложения доступен для всех запросов от всех клиентов (рис. 9.14). Он похож на одиночный бин. Различие состоит в том, что в данном случае нельзя создать в контексте несколько экземпляров. Кроме того, когда мы говорим о жизненном цикле бинов с областью видимости в веб-приложениях (включая область видимости в рамках всего веб-приложения), отправной точкой всегда являются HTTP-запросы. В случае бинов с областью

видимости в рамках приложения возникают те же проблемы конкурентности, которые были описаны в главе 5 для одиночных бинов. Желательно, чтобы атрибуты одиночных бинов были неизменяемыми. Тот же совет касается и бинов с областью видимости в рамках приложения. Но если сделать атрибуты неизменяемыми, вместо бина с областью видимости в рамках приложения можно просто использовать одиночный бин.

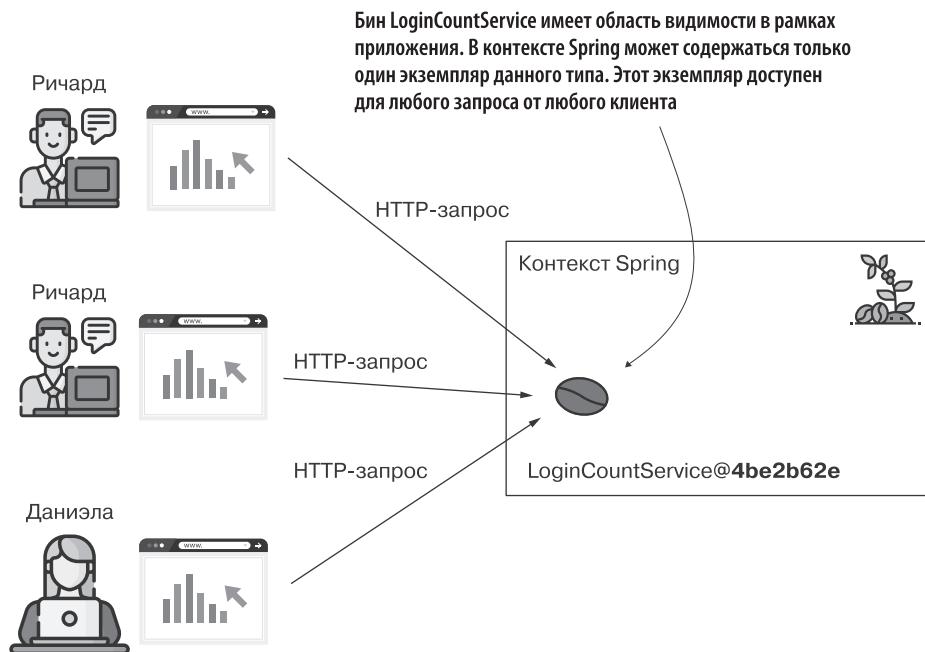


Рис. 9.14. Так выглядит бин с областью видимости в рамках всего веб-приложения Spring. Экземпляр этого бина доступен для всех HTTP-запросов от всех клиентов. В контексте Spring может существовать только один экземпляр бина данного типа, и его могут использовать все, кому он нужен

Обычно я рекомендую разработчикам по возможности не применять бины с областью видимости в рамках приложения. Лучше напрямую использовать уровень хранения данных, например базу данных (о которой вы узнаете в главе 11).

Для лучшего понимания чего-либо всегда лучше рассмотреть это на примере. Дополним приложение, с которым мы работали в данной главе, счетчиком попыток аутентификации. Этот пример находится в проекте sq-ch9-ex3.

Поскольку нам нужно подсчитать все попытки аутентификации от всех пользователей, мы будем хранить счетчик в бине с областью видимости в рамках приложения. Создадим такой бин LoginCountService и разместим счетчик в его атрибуте. Определение этого класса показано в листинге 9.12.

Листинг 9.12. Класс LoginCountService для подсчета попыток аутентификации

```

@Service
@ApplicationScope ←
public class LoginCountService {
    private int count;

    public void increment() {
        count++;
    }

    public int getCount() {
        return count;
    }
}

```

Аннотация @ApplicationScope распространяет область видимости бина на все приложение

Этот бин может автомонтируться в бин LoginProcessor, который может вызывать метод `increment()` при каждой попытке аутентификации, как показано в листинге 9.13.

Листинг 9.13. Увеличение счетчика попыток аутентификации при каждом запросе на аутентификацию

```

@Component
@RequestScope
public class LoginProcessor {

    private final LoggedUserManagementService loggedUserManagementService;
    private final LoginCountService loginCountService;

    private String username;
    private String password; ←
    public LoginProcessor( ←
        LoggedUserManagementService loggedUserManagementService,
        LoginCountService loginCountService) {
        this.loggedUserManagementService = loggedUserManagementService;
        this.loginCountService = loginCountService;
    }

    public boolean login() {
        loginCountService.increment(); ←
        String username = this.getUsername();
        String password = this.getPassword();

        boolean loginResult = false;
        if ("natalie".equals(username) && "password".equals(password)) {
            loginResult = true;
            loggedUserManagementService.setUsername(username);
        }
    }
}

```

Внедряем бин LoginCountService через параметры конструктора

Увеличиваем счетчик при каждой попытке аутентификации

```

        return loginResult;
    }

    // Остальной код
}

```

Наконец, нам осталось последнее: вывести это значение на экран. Как вы уже знаете из рассмотренных ранее примеров (начиная с главы 7), чтобы передать в представление значение счетчика, нужно использовать параметр `Model` в действии контроллера. Затем можно вывести это значение в представлении с помощью Thymeleaf. В листинге 9.14 показано, как отправить значение из контроллера в представление.

Листинг 9.14. Передача значения счетчика из контроллера для отображения на главной странице

```

@Controller
public class MainController {

    // Остальной код

    @GetMapping("/main")
    public String home(
        @RequestParam(required = false) String Logout,
        Model model
    ) {
        if (Logout != null) {
            loggedUserManagementService.setUsername(null);
        }

        String username = loggedUserManagementService.getUsername();
        int count = loginCountService.getCount(); ←
        if (username == null) { ←
            return "redirect:/";
        }

        model.addAttribute("username", username);
        model.addAttribute("loginCount", count); ←
        return "main.html";
    }
}

```

Получаем значение счетчика из бина с областью видимости в рамках приложения

Передаем значение счетчика в представление

В листинге 9.15 показано, как вывести значение счетчика на странице.

Листинг 9.15. Вывод значения счетчика на главной странице

```

<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>

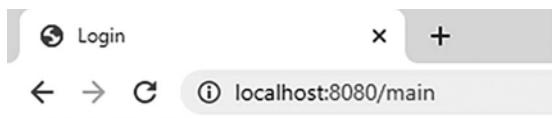
```

```

<meta charset="UTF-8">
<title>Login</title>
</head>
<body>
    <h1>Welcome, <span th:text="${username}"></span>!</h1>
    <h2>
        Your login number is
        <span th:text="${loginCount}"></span> ←———— Выводим значение счетчика на странице
    </h2>
    <a href="/main?Logout">Log out</a>
</body>
</html>

```

При запуске приложения вы обнаружите, что на главной странице появилась запись с общим количеством попыток аутентификации (рис. 9.15).



Welcome, natalie!

Your login number is 5

[Log out](#)

Рис. 9.15. Результат работы приложения — веб-страница, на которой отображается общее количество попыток аутентификации для всех пользователей. Это число выводится на главной странице

РЕЗЮМЕ

- Кроме одиночной и прототипной (рассмотренных в главах 2–5), в веб-приложениях Spring доступны еще три области видимости бинов. Они применимы только в веб-приложениях, и поэтому их называют областями веб-видимости:
 - *область видимости в рамках запроса* — Spring создает новый экземпляр бина для каждого HTTP-запроса;
 - *область видимости в рамках сессии* — Spring создает новый экземпляр бина для каждой HTTP-сессии конкретного клиента. Этот экземпляр доступен для всех запросов данного клиента в рамках данной сессии;

- *область видимости в рамках приложения* — во всем приложении может существовать только один экземпляр такого бина. Этот экземпляр доступен для всех запросов от всех клиентов.
- Spring гарантирует, что экземпляр бина с областью видимости в рамках запроса доступен только для данного HTTP-запроса. Поэтому можно смело использовать его атрибуты, не беспокоясь о проблемах конкурентности. Можно также не думать о том, что такие экземпляры займут всю память приложения: их время жизни очень коротко и они попадают в сборку мусора сразу же после завершения выполнения HTTP-запроса.
- Spring создает экземпляры бинов с областью видимости в рамках запроса для каждого HTTP-запроса, что происходит весьма часто. Поэтому не рекомендуется усложнять процесс за счет использования логики в конструкторе или в методе `@PostConstruct`.
- Экземпляры бинов с областью видимости в рамках сессии Spring связывает с конкретной HTTP-сессией клиента. Таким образом, их можно использовать для совместного доступа к данным для нескольких HTTP-запросов от одного и того же клиента.
- Но даже один и тот же клиент может отправлять конкурентные HTTP-запросы. Если такие запросы изменяют данные, хранящиеся в экземпляре бина с областью видимости в рамках сессии, это может привести к состоянию гонки. Необходимо учитывать возможность подобных ситуаций и либо избегать их, либо синхронизировать код, чтобы поддерживать конкурентность.
- Рекомендую по возможности не использовать бины с областью видимости в рамках приложения. Такие бины доступны для всех запросов веб-приложения, поэтому любая операция записи, скорее всего, будет нуждаться в синхронизации. Это приведет к образованию узких мест и значительно снизит производительность приложения. Более того, такие бины хранятся в памяти приложения в течение всего времени жизни приложения и не попадают в сборку мусора. Как вы узнаете в главе 11, лучше хранить данные непосредственно в базе данных.
- При использовании бинов с областями видимости в рамках сессии и в рамках приложения запросы становятся менее независимыми. В подобных случаях говорят, что приложение управляет состоянием, необходимым для запросов (или что это приложение с сохранением состояния). Приложениям с сохранением состояния свойственны различные проблемы архитектуры, которых лучше избегать. Описание этих проблем выходит за рамки нашей книги, но вам стоит заранее знать о них, чтобы сразу поискать альтернативы.

10

Реализация REST-сервисов

В этой главе

- ✓ REST-сервисы.
- ✓ Создание конечных точек REST.
- ✓ Управление данными, которые сервер передает в ответ на HTTP-запрос.
- ✓ Извлечение данных из тела HTTP-запроса, полученного от клиента.
- ✓ Обработка исключений на уровне конечной точки.

В главах 7–9, говоря о веб-приложениях, я несколько раз упоминал сервисы REST (Representational State Transfer — передача состояния представления). В этой главе мы займемся REST-сервисами вплотную, и вы узнаете, что они имеют отношение не только к веб-приложениям.

REST-сервисы — один из наиболее часто встречающихся способов построения коммуникации между приложениями. REST-сервисы обеспечивают доступ к функциональности сервера через конечные точки, к которым обращается клиент.

REST-сервисы применяют для установления соединения между клиентом и сервером в веб-приложении. Но их также можно использовать для организации обмена данными между мобильным приложением и бэкендом или даже между двумя сервисами бэкенда (рис. 10.1).

Конечная точка REST позволяет установить связь между двумя приложениями: одно приложение дает другому возможность пользоваться своим функционалом, делая его доступным через HTTP

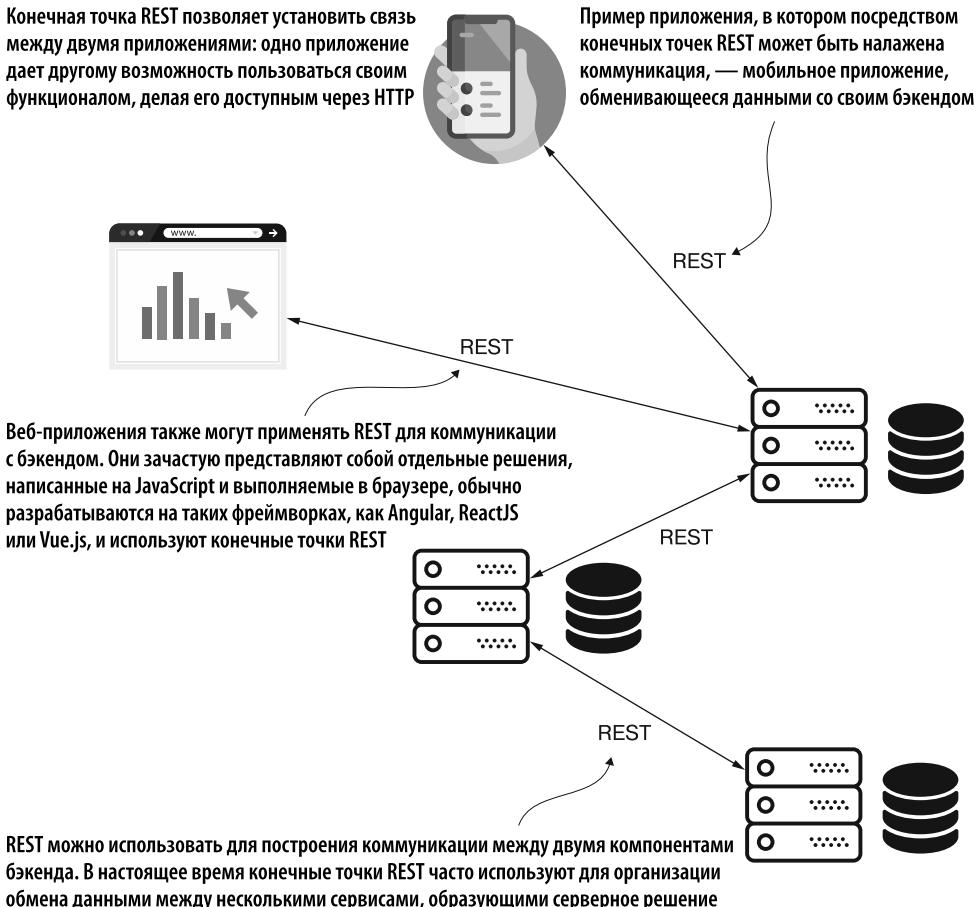


Рис. 10.1. REST-сервисы — это способ обмена данными между двумя приложениями. В настоящее время они применяются в самых разных областях. Обращаться к бэкенду с помощью конечных точек REST может не только клиент веб-приложения или мобильное приложение. Сервисы бэкенда также могут обмениваться данными посредством REST-вызовов

Поскольку во многих современных Spring-приложениях вам, скорее всего, встретятся сервисы REST и вам придется с ними работать, я решил, что изучение данной темы является обязательным для любого Spring-разработчика.

Прежде всего в разделе 10.1 мы выясним, что именно представляют собой REST-сервисы. Вы узнаете, что Spring поддерживает REST-сервисы посредством того же механизма Spring MVC, о котором мы говорили в главах 7–9. В разделе 10.2 мы рассмотрим основные синтаксические конструкции, которыми нужно владеть для работы с конечными точками REST. Мы выполним несколько

примеров, чтобы усвоить важнейшие моменты, обязательные для понимания любому Spring-разработчику при построении коммуникации между двумя приложениями посредством REST-сервисов.

10.1. ОБМЕН ДАННЫМИ МЕЖДУ ПРИЛОЖЕНИЯМИ ПОСРЕДСТВОМ REST-СЕРВИСОВ

Далее мы обсудим REST-сервисы и то, каким образом они реализованы в Spring посредством Spring MVC. Конечные точки REST — это всего лишь способ организовать коммуникацию между двумя приложениями. Они представляют собой обычное действие контроллера, связанное с HTTP-методом и путем. Приложение вызывает действие контроллера через HTTP. Поскольку таким образом приложение делает сервис доступным через веб-протокол, конечные точки можно назвать веб-сервисами.

В итоге в Spring конечная точка REST — это тоже действие контроллера, связанное с HTTP-методом и путем. Чтобы предоставить доступ к конечным точкам REST, Spring использует тот же известный вам механизм, что и для других веб-приложений. Однако теперь мы настроим диспетчер серверов Spring MVC так, чтобы он не искал представление для REST-сервисов. В диаграмме Spring MVC, которую вы помните из главы 7, в случае REST-сервисов пропадает арбитр представлений. В ответ на HTTP-запрос сервер возвращает клиенту ровно то, что он получил от действия контроллера. Эти изменения в процессе Spring MVC показаны на рис. 10.2.

Вы скоро обнаружите, что REST-сервисы очень удобны. Одна из причин их популярности — простота, а благодаря Spring создавать REST-сервисы стало и того проще. Но прежде, чем приступить к первому примеру, хочу предупредить вас о некоторых проблемах обмена данными, которые могут возникнуть при использовании конечных точек REST:

- если действие контроллера выполняется долго, то HTTP-вызов конечной точки может завершиться и соединение разорвется;
- при попытке передать большое количество данных за один вызов (один HTTP-запрос) времени вызова может не хватить и соединение разорвется. Передача за один REST-вызов больше пары мегабайт обычно не лучшее решение;
- слишком много конкурентных обращений к одной конечной точке бэкенда может привести к чрезмерной нагрузке на приложение и вызвать его сбой;
- HTTP-вызовы выполняются за счет сети, но сеть никогда не бывает абсолютно надежной. Всегда есть вероятность того, что вызов конечной точки REST завершится неудачно из-за сетевого сбоя.

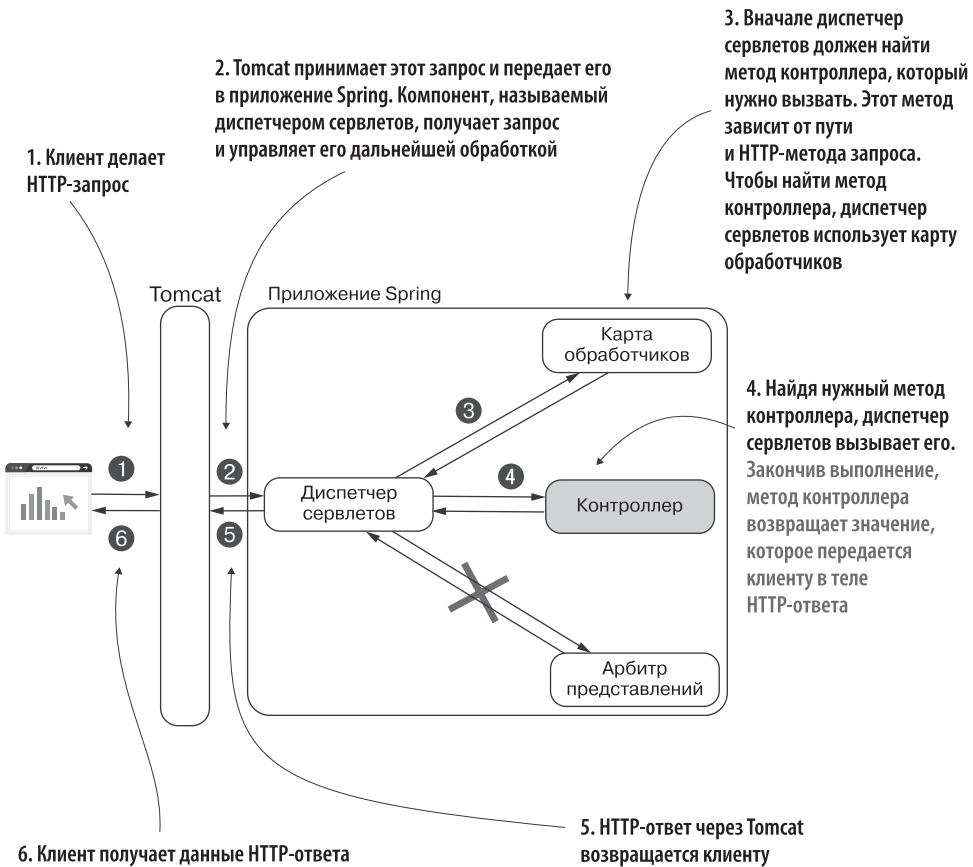


Рис. 10.2. При реализации конечных точек REST процесс Spring MVC выполняется немного по-другому. Приложению больше не нужен арбитр представлений, так как клиенту возвращаются данные, полученные непосредственно от действия контроллера. После завершения действия контроллера диспетчер сервлетов не формирует представление, а сразу передает HTTP-ответ

При построении коммуникации между двумя приложениями посредством REST всегда следует помнить о том, что может случиться при неудачном вызове и как это повлияет на работу приложения. Спросите себя: что будет с данными, если такое произойдет? Выдержит ли архитектура приложения несогласованность данных, если будет сбой при вызове конечной точки? Если приложение должно сообщить пользователю об ошибке, как именно это будет сделано? Все это весьма сложные вопросы. Чтобы на них ответить, нужны знания в области архитектуры приложений, которые выходят за рамки нашего издания. Но я советую вам прочитать книгу Дж. Дж. Гивакса «Паттерны проектирования API» (Питер, 2023) — отличное руководство, где приводятся лучшие рекомендации по разработке API.

10.2. СОЗДАНИЕ КОНЕЧНОЙ ТОЧКИ REST

Научимся создавать конечные точки REST с помощью Spring. Хорошая новость состоит в том, что для этого в Spring используется все тот же механизм Spring MVC, так что большую часть процесса вы уже знаете из глав 7 и 8. Для начала рассмотрим пример (проект sq-ch10-ex1). Возьмем за основу тот пример, который мы выполнили в главах 7 и 8, и подумаем, как можно преобразовать простой веб-контроллер в REST-контроллер, выполняющий веб-сервисы REST.

В листинге 10.1 показан класс контроллера, в котором реализовано простое действие. Как вы знаете из главы 7, класс контроллера сопровождается стереотипной аннотацией `@Controller`. Благодаря ей экземпляр такого класса становится бином в контексте Spring, и Spring MVC знает, что это контроллер, методы которого связаны с определенными путями HTTP. Мы также воспользовались аннотацией `@GetMapping`, чтобы указать путь и HTTP-метод для действия. Единственный новый элемент, который вам встретится в этом листинге, — аннотация `@ResponseBody`. Она сообщает диспетчеру сервлетов, что действие контроллера не возвращает имя представления, а передает данные непосредственно в HTTP-запрос.

Листинг 10.1. Класс контроллера, содержащий действие для конечной точки REST

```
@Controller
public class HelloController {
    @GetMapping("/hello")
    @ResponseBody
    public String hello() {
        return "Hello!";
    }
}
```

С помощью аннотации `@Controller` обозначаем, что данный класс является контроллером Spring MVC

С помощью аннотации `@GetMapping` связываем HTTP-метод GET и путь с действием контроллера

С помощью аннотации `@ResponseBody` сообщаем диспетчеру сервлетов, что этот метод возвращает не имя представления, а непосредственно HTTP-ответ

А теперь посмотрим, что произойдет, если добавить в контроллер еще несколько методов, как показано в листинге 10.2. Постоянное применение аннотации `@ResponseBody` к каждому методу выглядит раздражающее.

Листинг 10.2. Применение аннотации `@ResponseBody`, которое приводит к дублированию кода

```
@Controller
public class HelloController {
    @GetMapping("/hello")
    @ResponseBody
    public String hello() {
        return "Hello!";
    }
}
```

```

@GetMapping("/ciao")
@ResponseBody
public String ciao() {
    return "Ciao!";
}
}

```

Дублирования кода лучше избегать. Желательно как-нибудь избавиться от повторения `@ResponseBody` для каждого метода. Чтобы решить эту проблему, в Spring есть аннотация `@RestController`, которая заменяет сочетание `@Controller` и `@ResponseBody`. С помощью `@RestController` мы сообщаем Spring, что все действия контроллера являются конечными точками REST, поэтому в много-кратном использовании аннотации `@ResponseBody` нет необходимости. Чтобы протестировать и сравнить оба варианта, я вынес этот код в отдельный проект `sq-ch10-ex2` (листинг 10.3).

Листинг 10.3. Аннотация `@RestController`, позволяющая избавиться от дублирования кода

```

@RestController ←
public class HelloController {

    @GetMapping("/hello")
    public String hello() {
        return "Hello!";
    }

    @GetMapping("/ciao")
    public String ciao() {
        return "Ciao!";
    }
}

```

Вместо повторения аннотации
`@ResponseBody` перед каждым
методом, заменяем `@Controller`
на `@RestController`

Создать пару конечных точек очень легко, не правда ли? Но как проверить, правильно ли они работают? Ниже вы освоите два инструмента, позволяющих обращаться к конечным точкам, которые часто применяются в реальных приложениях:

- *Postman* — имеет приятный GUI и удобен в использовании;
- *cURL* — инструмент командной строки, пригодный в тех случаях, когда нельзя использовать GUI (например, при соединении с виртуальной машиной по SSH или если вы пишете сценарий пакетной обработки).

Оба эти инструмента обязательно должен знать любой разработчик. В главе 15 вы познакомитесь с третьим способом, позволяющим убедиться в правильности работы конечных точек, — интеграционными тестами.

Для начала запустим приложение — проект `sq-ch10-ex1` либо `sq-ch10-ex2` — выбор не имеет значения, их поведение одинаково. Единственное различие состоит

в синтаксисе. Как вы уже знаете из главы 7, по умолчанию приложение Spring Boot настраивает контейнер сервлетов Tomcat таким образом, что он доступен через порт 8080.

Начнем с Postman. Вам нужно установить этот инструмент на своем компьютере согласно инструкциям, которые находятся на официальном сайте <https://www.postman.com/>. После установки Postman откройте его. Вы увидите интерфейс, подобный тому, что показан на рис. 10.3.

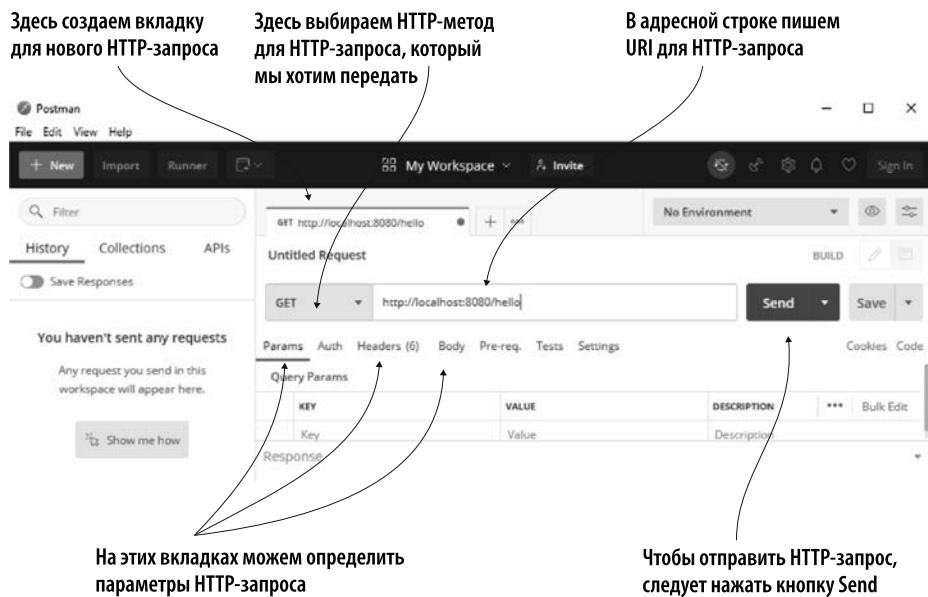


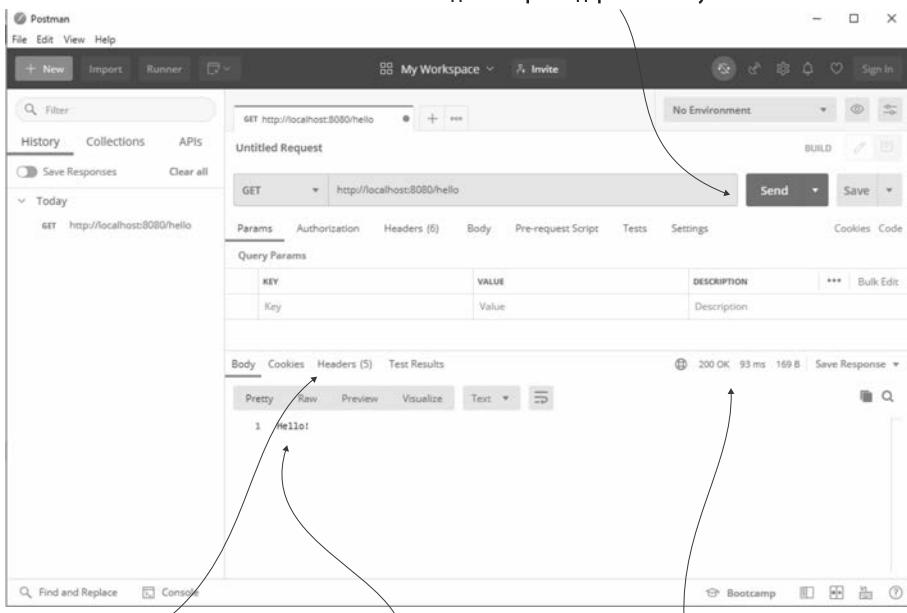
Рис. 10.3. Postman предоставляет удобный интерфейс для настройки параметров и передачи HTTP-запросов. Чтобы отправить HTTP-запрос, нужно выбрать HTTP-метод, указать URI запроса и нажать кнопку Send. При необходимости можно задать и другие элементы конфигурации — параметры запроса, заголовки, тело запроса

Когда вы нажмете кнопку Send, Postman отправит HTTP-запрос. Когда он будет выполнен, Postman выведет содержимое HTTP-ответа, как показано на рис. 10.4.

Если у вас нет GUI, для вызова конечной точки можно использовать инструмент командной строки. Читая книги и статьи, вы обнаружите, что в них для демонстрации вместо приложений с GUI обычно применяется именно этот способ, поскольку с ним можно показать команды более лаконично.

Если в качестве инструмента командной строки вы решите использовать cURL, то вам, как и в случае с Postman, вначале понадобится установить этот инструмент. Процедура установки cURL зависит от используемой операционной системы и описана на официальном сайте <https://curl.se/>.

После того как вы нажмете кнопку Send, Postman отправит HTTP-запрос. Когда HTTP-запрос будет выполнен, Postman выведет на экран содержимое полученного HTTP-ответа



Если в HTTP-ответе есть заголовки, они появятся на этой вкладке Postman

Здесь выводится тело HTTP-ответа. В данном случае это строка Hello!

Здесь вы найдете код статуса HTTP-ответа, а также время его выполнения и количество переданных данных (в байтах)

Рис. 10.4. Когда HTTP-запрос выполнен, Postman выводит содержимое HTTP-ответа. Здесь вы найдете статус HTTP-ответа, время, затраченное на выполнение запроса, объем переданных данных в байтах, а также тело и заголовки ответа

После установки и настройки инструмента, можно передавать HTTP-запросы с помощью команды `curl`. В следующей строке кода показана команда, позволяющая отправить HTTP-запрос, чтобы проверить конечную точку `/hello`, созданную в нашем приложении:

```
curl http://localhost:8080/hello
```

Когда HTTP-запрос будет выполнен, в консоль будет выведено только тело HTTP-ответа:

Hello!

Если в запросе используется HTTP-метод GET, его можно не указывать явно. Если же это какой-либо другой метод или если вы все же хотите четко прописать GET, нужно воспользоваться флагом `-X`:

```
curl -X GET http://localhost:8080/hello
```

Чтобы получить другую информацию о HTTP-ответе, добавьте в команду флаг `-v`:

```
curl -v http://localhost:8080/hello
```

В следующем фрагменте показан результат выполнения этой команды — как видите, он более развернутый. В этом длинном тексте вы найдете информацию о статусе, количестве отправленных данных и заголовки:

```
Trying ::1:8080...
* Connected to localhost (::1) port 8080 (#0)
> GET /hello HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.73.0
> Accept: */*
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 ← Статус HTTP-ответа
< Content-Type: text/plain; charset=UTF-8
< Content-Length: 6
< Date: Fri, 25 Dec 2020 23:11:02 GMT
<
{ [6 bytes data]
100   6   100   6   0   0    857   0 --::--- --::--- --::--- --::---
1000
Hello! ← Тело HTTP-ответа
* Connection #0 to host localhost left intact
```

10.3. УПРАВЛЕНИЕ HTTP-ОТВЕТОМ

В этом разделе вы узнаете, как управлять HTTP-ответом в действии контроллера. HTTP-ответ — это тот способ, которым серверная часть приложения возвращает данные клиенту в ответ на его запрос. В HTTP-ответе содержится следующая информация:

- *заголовки ответа* — небольшие фрагменты данных (как правило, не длиннее пары слов);
- *тело ответа* — более крупный объем данных, которые бэкенд должен передать в ответ на запрос клиента;
- *статус ответа* — краткое представление результата запроса.

Прежде чем двигаться дальше, потратьте пару минут, чтобы бегло ознакомиться с приложением B и освежить в памяти детали HTTP-протокола. Подразделы 10.3.1 и 10.3.2 будут посвящены разным вариантам передачи данных в теле HTTP-ответа. В подразделе 10.3.3 вы узнаете, как при необходимости можно передать статус и заголовки HTTP-ответа.

10.3.1. Передача объектов в теле HTTP-ответа

Рассмотрим передачу экземпляров объектов в теле HTTP-ответа. Чтобы отправить объект клиенту в HTTP-ответе, нужно всего лишь сделать так, чтобы действие контроллера возвращало этот объект. В примере sq-ch10-ex3 мы создадим объект-модель **Country** с атрибутами **name** (название страны) и **population** (число жителей в миллионах человек) и пропишем действие контроллера, которое будет возвращать экземпляр типа **Country**.

В листинге 10.4 показан класс, определяющий объект **Country**. При использовании объекта (такого как **Country**) в качестве модели данных, передаваемых между двумя приложениями, его принято называть **DTO** (data transfer object — объект передачи данных). Нашим **DTO** является объект **Country**, экземпляры которого созданная конечная точка REST будет возвращать в теле HTTP-ответа.

Листинг 10.4. Модель данных, возвращаемых сервером в теле HTTP-ответа

```
public class Country {

    private String name;
    private int population;

    public static Country of( ←
        String name,
        int population) {
        Country country = new Country();
        country.setName(name);
        country.setPopulation(population);
        return country;
    }

    // Геттеры и сеттеры
}
```

Чтобы было проще, мы определили для экземпляра **Country** статический метод генерации объекта, который принимает название страны и количество жителей и возвращает экземпляр **Country** с указанными значениями

В листинге 10.5 показано, как выглядит действие контроллера, которое возвращает экземпляр типа **Country**.

Листинг 10.5. Действие контроллера, возвращающее экземпляр объекта

```
@RestController ←
public class CountryController {

    @GetMapping("/france") ←
    public Country france() {
        Country c = Country.of("France", 67);
        return c; ← Возвращаем экземпляр типа Country
    }
}
```

Обозначаем класс как REST-контроллер, чтобы Spring добавил бин в контекст и чтобы диспетчер сервлетов не искал представление после окончания работы этого метода

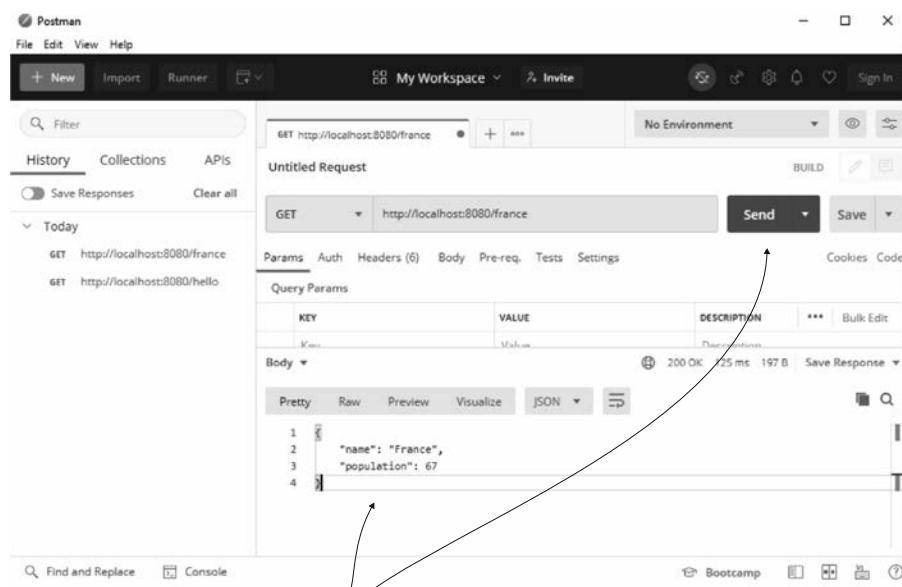
Связываем действие контроллера с HTTP-методом GET и путем /france

Что произойдет при вызове конечной точки? Как будет выглядеть объект в теле HTTP-ответа? По умолчанию Spring создает строку, представляющую объект, и дает ее в формате JSON. JavaScript Object Notation (JSON) — это простой способ представления строк в виде пар «атрибут — значение». Скорее всего, записи в данном формате вам уже встречались. Но если вам еще не приходилось их использовать, я подготовил описание всего, что может понадобиться, и вынес в приложение Г.

При вызове конечной точки /france тело ответа выглядит так:

```
{
    "name": "France",
    "population": 67
}
```

Рисунок 10.5 напомнит вам, где искать тело HTTP-ответа при вызове конечной точки из Postman.



Здесь вы найдете тело HTTP-ответа,
после того как отправите запрос, нажав Send

Рис. 10.5. Когда вы нажмете кнопку Send, Postman отправит запрос. Когда он будет выполнен, Postman выведет информацию об ответе, в том числе тело ответа

В теле ответа можно также передавать экземпляры, представляющие собой коллекции объектов. В листинге 10.6 показан новый метод, возвращающий список объектов Country.

Листинг 10.6. Возвращение коллекции в теле ответа

```

@RestController
public class CountryController {

    // Остальной код

    @GetMapping("/all")
    public List<Country> countries() {
        Country c1 = Country.of("France", 67);
        Country c2 = Country.of("Spain", 47);

        return List.of(c1,c2); ← Возвращает коллекцию в теле HTTP-ответа
    }
}

```

При вызове конечной точки тело ответа выглядит так:

```

[ ← В JSON список заключается в квадратные скобки
{
    "name": "France",
    "population": 67
},
{
    "name": "Spain",
    "population": 47
}
]

```

Каждый объект заключается в фигурные скобки, объекты перечисляются через запятую

JSON — наиболее распространенный способ представления объектов при работе с конечными точками REST. Вы не обязаны использовать именно его, но вам едва ли встретится кто-либо, кто применяет что-то другое. При желании вы можете использовать в Spring другие форматы для тела запроса (такие как XML или YAML), подключив к объектам специальный преобразователь. Но вероятность того, что вам это понадобится на практике, столь мала, что мы пропустим обсуждение данной темы и перейдем к следующему важному вопросу, который стоит изучить.

10.3.2. Создание HTTP-ответа со статусом и заголовками

Сосредоточим наше внимание на статусе и заголовках HTTP-ответа. Иногда бывает удобнее передать часть данных в заголовке ответа. Статус ответа — это важный признак HTTP-ответа, который сообщает о результате запроса. По умолчанию Spring присваивает ответам следующие основные HTTP-статусы:

- 200 — OK — при обработке запроса на стороне сервера не возникло никаких исключений;

- 404 — Not Found — запрошенный ресурс не существует;
- 400 — Bad Request — часть запроса не соответствует тем данным, которые ожидает сервер;
- 500 — Error on server — при обработке запроса на стороне сервера по какой-то причине возникло исключение. Как правило, с подобным исключением клиент ничего не может сделать. Предполагается, что проблема будет решена на стороне бэкенда.

Но иногда требования, предъявляемые к приложению, вынуждают создать специальный статус. Как это сделать? Самый простой и наиболее распространенный способ изменить HTTP-ответ — применить класс `ResponseEntity`.

Этот класс Spring позволяет модифицировать тело, статус и заголовки HTTP-ответа. Использование `ResponseEntity` показано в проекте sq-ch10-ex4. В листинге 10.7 действие контроллера вместо объекта, который должен быть помещен непосредственно в тело HTTP-ответа, возвращает экземпляр `ResponseEntity`.

Класс `ResponseEntity` позволяет определить не только тело HTTP-ответа, но также статус и заголовки. В листинге мы создадим три заголовка и изменим статус HTTP-ответа на 202 — Accepted.

Листинг 10.7. Создание HTTP-ответа со специальным статусом и заголовками

```
@RestController
public class CountryController {

    @GetMapping("/france")
    public ResponseEntity<Country> france() {
        Country c = Country.of("France", 67);
        return ResponseEntity
            .status(HttpStatus.ACCEPTED) ← Меняем статус HTTP-ответа
            .header("continent", "Europe") ← на 202 Accepted
            .header("capital", "Paris") ← Добавляем к HTTP-ответу
            .header("favorite_food", "cheese and wine") ← три дополнительных
            .body(c); ← Создаем тело HTTP-ответа
    }
}
```

Если вы отправляли HTTP-запрос из Postman, можете убедиться, что статус HTTP-ответа изменился на 202 Accepted, как показано на рис. 10.6.

На вкладке Headers в Postman вы также увидите три созданных нами новых заголовка HTTP-ответа (рис. 10.7).

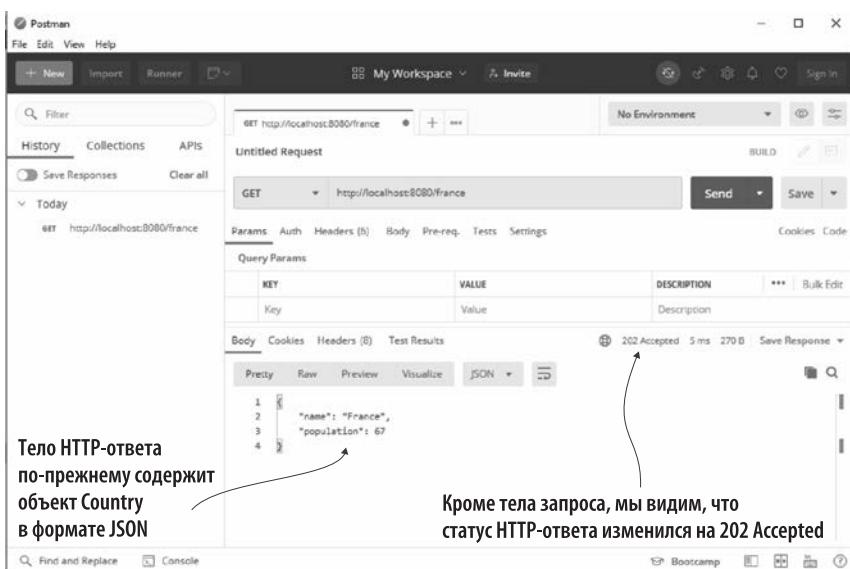


Рис. 10.6. Нажав кнопку Send и отправив HTTP-запрос, а затем получив HTTP-ответ, видим, что статус ответа изменился на 202 Accepted. Тело ответа по-прежнему представляет собой строку в формате JSON

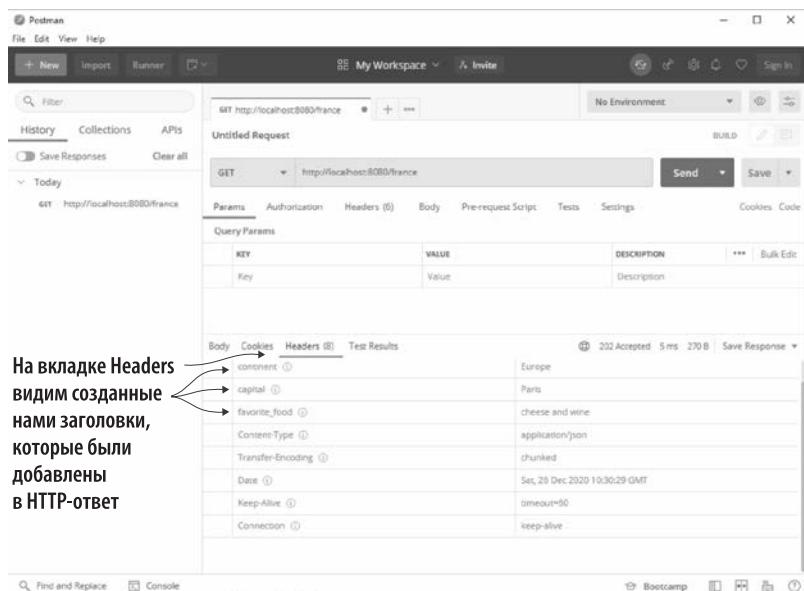


Рис. 10.7. Чтобы увидеть новые заголовки HTTP-ответа в Postman, перейдите на вкладку Headers

10.3.3. Управление исключениями на уровне конечной точки

Очень важно продумать, что случится, если действие контроллера выбросит исключение. Мы часто используем исключения, чтобы сообщать о каких-то специфических ситуациях, часть из которых связана с бизнес-логикой приложения. Предположим, мы создали конечную точку, через которую клиенты проводят платежи. Если у пользователя недостаточно денег на счету, приложение может оповестить об этом, выдав исключение. В данном случае вы, очевидно, захотите добавить в HTTP-ответ некоторую информацию, чтобы рассказать клиенту о ситуации подробнее.

Один из способов управления исключениями состоит в том, чтобы перехватывать их в действии контроллера и, используя класс `ResponseEntity`, с которым вы познакомились в подразделе 10.3.2, передавать другую конфигурацию ответа в случае возникновения исключения.

Вначале рассмотрим пример с использованием этой методики, а затем я покажу вам другой вариант, который предпочитаю применять сам в сочетании с классом совета, реализующим REST-контроллер: аспект, который перехватывает вызов конечной точки, если она выдает исключение. В этом случае можно выполнять свою логику для каждого такого исключения.

Создадим проект с именем `sq-ch10-ex5`. В нашем примере мы определим исключение `NotEnoughMoneyException`. Приложение будет выдавать его при невозможности выполнить платеж, когда на счету клиента недостаточно денег. Определение класса исключения выглядит так:

```
public class NotEnoughMoneyException extends RuntimeException {  
}
```

Мы также создадим класс сервиса, в котором будет реализован сценарий использования. В данном примере он просто выдаст исключение. В реальных приложениях такие сервисы выполняют сложную логику электронных платежей. Класс сервиса, используемый в нашем примере, выглядит так:

```
@Service  
public class PaymentService {  
  
    public PaymentDetails processPayment() {  
        throw new NotEnoughMoneyException();  
    }  
}
```

Метод `processPayment()` возвращает значение типа `PaymentDetails`. `PaymentDetails` — это просто класс модели, описывающий тело HTTP-ответа,

который действие контроллера будет передавать в случае успешного платежа. Класс `PaymentDetails` выглядит так:

```
public class PaymentDetails {  
  
    private double amount;  
  
    // Геттеры и сеттеры  
}
```

Когда в приложении появляется исключение, используется другой класс модели, `ErrorDetails`, который сообщает клиенту о возникшей ситуации. Класс `ErrorDetails` тоже очень простой — он содержит всего один атрибут с уведомлением об ошибке и выглядит так:

```
public class ErrorDetails {  
  
    private String message;  
  
    // Геттеры и сеттеры  
}
```

Откуда контроллер знает, какой из объектов нужно вернуть, в зависимости от последовательности проведения операций? Если исключение не возникло (приложение успешно выполнило платеж), нужно вернуть HTTP-ответ со статусом `Accepted` и данными типа `PaymentDetails`. Предположим, при обработке запроса в приложении появилось исключение. В этом случае действие контроллера выдаст HTTP-ответ со статусом `400 Bad Request` и экземпляром `ErrorDetails`, в котором содержится сообщение с описанием проблемы. На рис. 10.8 наглядно показаны взаимосвязи между компонентами приложения и обязанности этих компонентов.

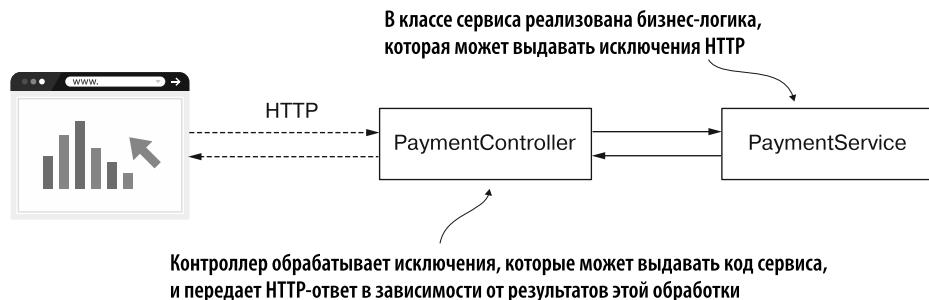


Рис. 10.8. В классе `PaymentService` реализована бизнес-логика, которая может выдавать исключения. Класс `PaymentController` обрабатывает эти исключения и по результатам действия отправляет клиенту HTTP-ответ

В листинге 10.8 показана логика, реализованная в методе контроллера.

Листинг 10.8. Обработка HTTP-ответа в действии контроллера в случае возникновения исключения

```
@RestController
public class PaymentController {

    private final PaymentService paymentService;

    public PaymentController(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    @PostMapping("/payment")
    public ResponseEntity<?> makePayment() {
        try {
            PaymentDetails paymentDetails = paymentService.processPayment();
            return ResponseEntity
                .status(HttpStatus.ACCEPTED)
                .body(paymentDetails);
        } catch (NotEnoughMoneyException e) {
            ErrorDetails errorDetails = new ErrorDetails();
            errorDetails.setMessage("Not enough money to make the payment.");
            return ResponseEntity
                .badRequest()
                .body(errorDetails);
        }
    }
}
```

Пытаемся вызвать метод сервиса `processPayment()`

В случае успешного завершения метода сервиса возвращаем HTTP-ответ со статусом Accepted и экземпляром `PaymentDetails`, содержащимся в теле ответа

Если выдано исключение типа `NotEnoughMoneyException`, передаем HTTP-ответ со статусом Bad Request и экземпляром `ErrorDetails`, содержащимся в теле ответа

Запустите приложение и вызовите конечную точку, используя Postman или curl. Мы заведомо сделали так, чтобы метод сервиса всегда возвращал `NotEnoughMoneyException`, так что ожидаем получить в HTTP-ответе сообщение о статусе 400 Bad Request, а в теле ответа — сообщение об ошибке. Результат отправки запроса в конечную точку /payment с помощью Postman показан на рис. 10.9.

Это хорошая методика, и многие разработчики ею пользуются для обработки исключений. Но в более сложных приложениях удобнее отделить управление исключениями от других обязанностей. Во-первых, иногда одно и то же исключение должно обрабатываться для нескольких конечных точек и, разумеется, мы не хотим, чтобы это привело к дублированию кода. Во-вторых, когда вам придется разбираться в работе тех или иных исключений, гораздо удобнее знать, что вся логика их обработки размещается в одном месте. Исходя из этих соображений, я предлагаю использовать совет REST-контроллера — аспект, который перехватывает исключения, выдаваемые действиями контроллера, и применяет к ним написанную вами логику в зависимости от конкретного случая.

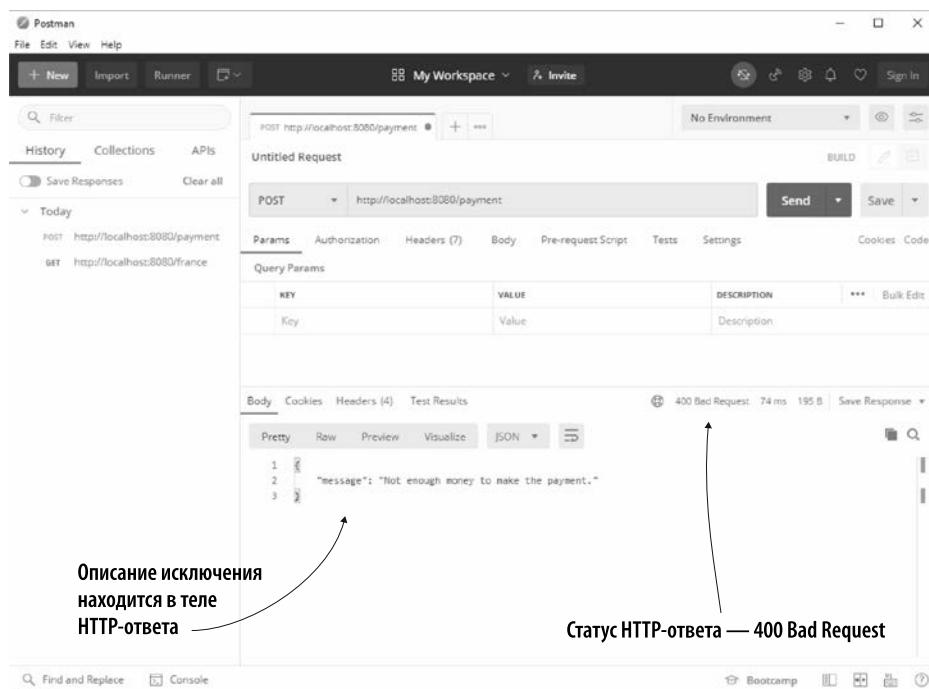


Рис. 10.9. Вызывая конечную точку /payment, получаем HTTP-ответ со статусом 400 Bad Request. В теле ответа находится сообщение об исключении

На рис. 10.10 показано, какие изменения нужно внести в структуру класса. Уделите некоторое время сравнению новой структуры класса с той, которая изображена на рис. 10.8.

Эти изменения реализованы в проекте sq-ch10-ex6. Как видно из листинга 10.9, действие контроллера стало гораздо проще благодаря тому, что оно больше не должно обрабатывать исключения.

Листинг 10.9. Действие контроллера, больше не обрабатывающее исключения

```

@RestController
public class PaymentController {

    private final PaymentService paymentService;

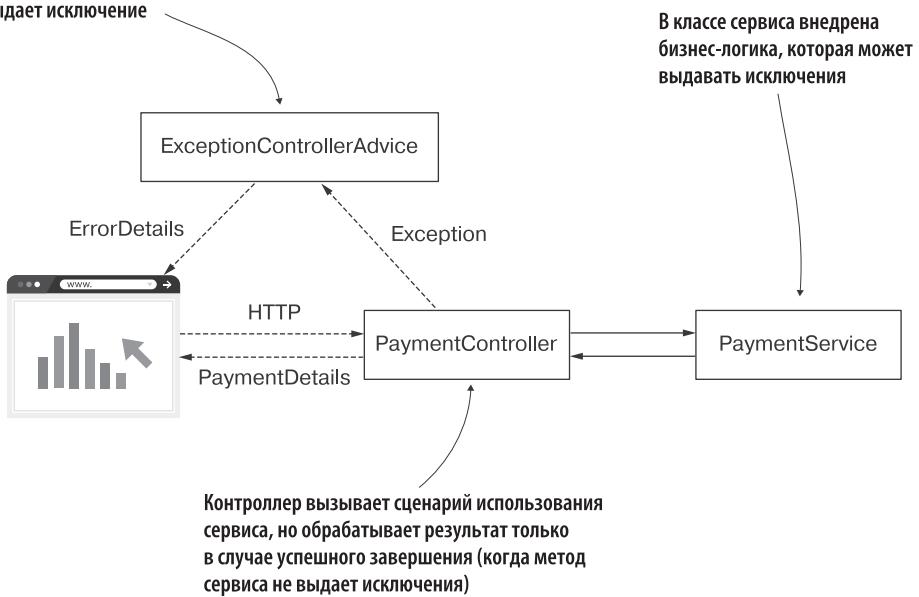
    public PaymentController(PaymentService paymentService) {
        this.paymentService = paymentService;
    }
}
        
```

```

@PostMapping("/payment")
public ResponseEntity<PaymentDetails> makePayment() {
    PaymentDetails paymentDetails = paymentService.processPayment();
    return ResponseEntity
        .status(HttpStatus.ACCEPTED)
        .body(paymentDetails);
}
}

```

В совете контроллера реализована логика для тех случаев, когда сервис выдает исключение



В классе сервиса внедрена бизнес-логика, которая может выдавать исключения

Рис. 10.10. Вместо того чтобы обрабатывать исключения, теперь контроллер занимается только ситуациями, когда сценарий использования завершился успешно. Для выполнения логики, созданной на случай, если действие контроллера выдаст исключение, мы добавили в контроллер совет `ExceptionControllerAdvice`

Взамен мы создали отдельный класс `ExceptionControllerAdvice`, в котором выполняется все, что нужно в случае, если действие контроллера выдает исключение `NotEnoughMoneyException`. Класс `ExceptionControllerAdvice` — это совет REST-контроллера. Чтобы отметить его как совет, мы воспользовались аннотацией `@RestControllerAdvice`. Метод, определенный в данном классе, называют также обработчиком исключений. Чтобы обозначить, какие именно исключения активируют выполнение этого метода, перед ним ставится аннотация `@ExceptionHandler`.

В листинге 10.10 показано определение класса совета REST-контроллера и метод-обработчик исключения, в котором реализована логика, связанная с исключением `NotEnoughMoneyException`.

Листинг 10.10. Логика обработки исключения, вынесенная в класс совета REST-контроллера

```
@RestControllerAdvice ←
public class ExceptionControllerAdvice { ← С помощью аннотации @RestControllerAdvice
    отмечаем класс как совет REST-контроллера
    @ExceptionHandler(NotEnoughMoneyException.class) ←
    public ResponseEntity<ErrorDetails> exceptionNotEnoughMoneyHandler() {
        ErrorDetails errorDetails = new ErrorDetails();
        errorDetails.setMessage("Not enough money to make the payment.");
        return ResponseEntity
            .badRequest()
            .body(errorDetails);
    }
}
```

Используем аннотацию @ExceptionHandler,
чтобы связать логику этого метода
с конкретным исключением

ПРИМЕЧАНИЕ

На практике вам иногда придется передавать из действия контроллера в совет дополнительную информацию о возникшем исключении. В таком случае нужно добавить параметр к методу-обработчику исключения в классе совета. Spring догадается, что нужно передать из контроллера в метод-обработчик ссылку на исключение. Затем можно будет использовать в логике совета любую информацию, извлеченную из экземпляра исключения.

10.4. ИЗВЛЕЧЕНИЕ ДАННЫХ ИЗ ТЕЛА ЗАПРОСА, ПОЛУЧЕННОГО ОТ КЛИЕНТА

Поговорим о том, как извлекать данные из тела HTTP-запроса, полученного от клиента. Как вы узнали в главе 8, мы можем передавать данные в HTTP-запросе, используя параметры запроса и переменные пути. Поскольку в основе конечных точек REST лежит все тот же механизм Spring MVC, синтаксис передачи данных через параметры запроса и переменные пути в этом случае ничем не отличается от уже известного вам. Конечные точки REST создаются точно так же, как и действия контроллера для веб-страниц; в обоих случаях используются одни и те же аннотации.

Но мы не затронули еще один важный вопрос: у HTTP-запроса есть тело и его можно использовать для передачи данных от клиента серверу. Тело HTTP-запроса часто используется при отправке информации в конечные точки REST. Как отмечается, в том числе в приложении B, если нужно передать большой

объем данных (по моему опыту, все, что занимает больше 50–100 символов), следует использовать тело запроса.

Чтобы передать данные в теле запроса, достаточно снабдить параметр действия контроллера аннотацией `@RequestBody`. По умолчанию Spring предполагает, что данные в параметре, сопровождаемом аннотацией, представлены в формате JSON, и пытается преобразовать строку JSON в объект, соответствующий типу параметра. Если фреймворку это не удается, приложение возвращает HTTP-ответ со статусом `400 Bad Request`. В проекте `sq-ch10-ex7` реализован простой пример с использованием тела HTTP-запроса. В контроллере определено действие, связанное с путем `/payment` и HTTP-методом POST. Действие пытается извлечь из тела запроса данные типа `PaymentDetails`. Контроллер выводит в консоль сервера сумму платежа, полученную из объекта `PaymentDetails`, и возвращает этот же объект клиенту в теле HTTP-ответа.

В листинге 10.11 показано определение контроллера из проекта `sq-ch10-ex7`.

Листинг 10.11. Извлечение данных из тела HTTP-запроса, полученного от клиента

```

@RestController
public class PaymentController {

    private static Logger logger =
        Logger.getLogger(PaymentController.class.getName());

    @PostMapping("/payment")
    public ResponseEntity<PaymentDetails> makePayment(
        @RequestBody PaymentDetails paymentDetails) { ←
        ↑ Извлекаем информацию
        ↑ о платеже из тела
        ↑ HTTP-запроса

        logger.info("Received payment " + ←
        paymentDetails.getAmount()); ←
        ↑ Выводим в консоль
        ↑ сервера сумму платежа

        return ResponseEntity {←
            .status(HttpStatus.ACCEPTED) ←
            .body(paymentDetails); ←
            ↑ Возвращаем объект с информацией
            ↑ о платеже в теле HTTP-ответа и присваиваем
            ↑ ответу статус 202 Accepted
        }
    }
}

```

На рис. 10.11 показано, как вызвать конечную точку `/payment` с телом запроса, используя Postman.

Если вы предпочитаете использовать cURL, можете использовать следующую команду:

```

curl -v -X POST http://127.0.0.1:8080/payment -d '{"amount": 1000}' -H
  "Content-Type: application/json"

```

Чтобы записать данные в тело запроса, откройте вкладку Body в конфигурации HTTP-запроса. Здесь включите режим raw, установив соответствующий переключатель, и выберите стиль форматирования JSON

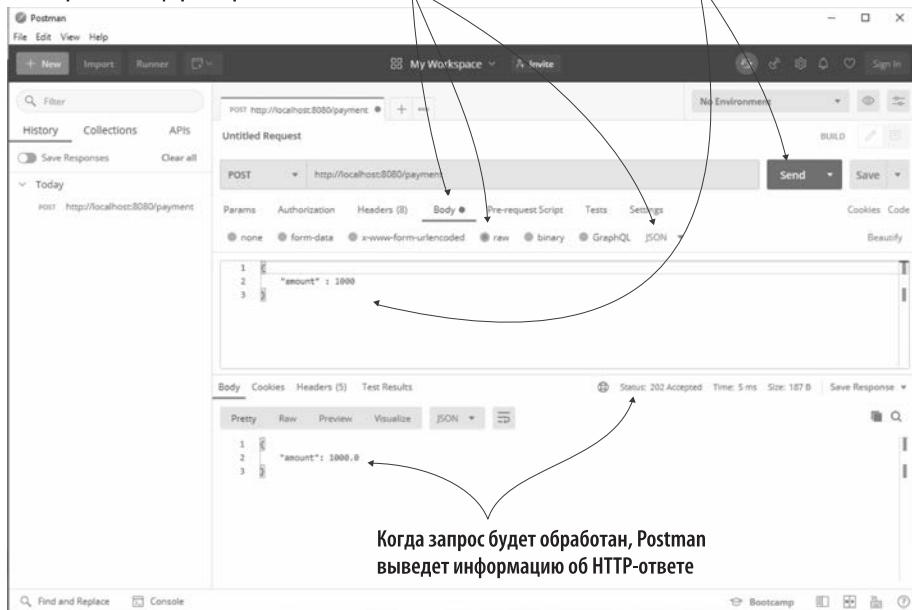


Рис. 10.11. Создание тела HTTP-запроса и вызов конечной точки с помощью Postman. Нужно ввести в текстовом поле ввода тело HTTP-запроса в формате JSON и указать, что эти данные закодированы в формате JSON. Когда HTTP-запрос будет обработан, в Postman появится информация об HTTP-ответе

МОЖНО ЛИ ИСПОЛЬЗОВАТЬ ТЕЛО ЗАПРОСА В СЛУЧАЕ HTTP-МЕТОДА GET?

Студенты часто задают мне этот вопрос. Почему применение тела запроса для HTTP-метода GET вызывает сомнения? До 2014 года спецификация HTTP-протокола не допускала использования тела запроса для вызовов HTTP GET. Не существовало реализаций для клиента и сервера, которые бы позволяли осуществить подобное.

В 2014 году спецификация изменилась, и теперь использовать тело запроса для HTTP-метода GET можно. Но иногда студентам попадаются в интернете статьи и книги в устаревших редакциях, и это приводит к затруднениям даже много лет спустя.

Подробнее об использовании HTTP-метода GET вы можете прочесть в подразделе 4.3.1 спецификации HTTP, RFC 7231: <https://tools.ietf.org/html/rfc7231#page-24>.

РЕЗЮМЕ

- Веб-сервисы REST (Representational State Transfer, передача состояния представления) — это простой способ организовать обмен данными между двумя приложениями.
- В Spring-приложениях конечные точки REST создаются на основе механизма Spring MVC. Необходимо либо использовать аннотацию `@ResponseBody`, чтобы обозначить, что данный метод возвращает непосредственно тело HTTP-ответа, либо заменить аннотацию `@Controller` на `@RestController`, чтобы создать конечную точку REST. Если не использовать какую-то из этих аннотаций, диспетчер сервлетов будет считать, что данный метод контроллера возвращает имя представления, и попытается найти это представление, вместо того чтобы передать HTTP-ответ.
- Можно создать действие контроллера, которое будет возвращать непосредственно тело HTTP-ответа, тогда Spring присвоит этому ответу статус по умолчанию.
- Чтобы изменить статус и заголовки HTTP-ответа, действие контроллера должно возвращать экземпляр `ResponseEntity`.
- Один из способов обработки исключений — перехватывать их непосредственно на уровне действия контроллера. Но иногда такой способ приводит к дублированию кода, чего лучше не допускать.
- Вместо того чтобы обрабатывать исключения непосредственно в контроллере, можно вынести логику, которая выполняется в случае, если действие контроллера выдает исключение, в класс совета REST-контроллера.
- Конечная точка может получать данные от клиента через HTTP-запрос посредством параметров запроса, переменных пути, а также из тела HTTP-запроса.

11

Использование конечных точек REST

В этой главе

- ✓ Вызов конечных точек REST с помощью OpenFeign из проекта Spring Cloud.
- ✓ Вызов конечных точек REST с помощью RestTemplate.
- ✓ Вызов конечных точек REST с помощью WebClient.

В главе 10 мы рассмотрели создание конечных точек REST. REST-сервисы — типичный способ организации обмена данными между двумя системными компонентами. С помощью REST-сервисов клиент веб-приложения может обратиться к бэкенду или же один из компонентов бэкенда — к другому. В серверных решениях, состоящих из нескольких сервисов (см. приложение А), таким компонентам нужен способ «поговорить друг с другом» и обменяться данными. Поэтому при реализации подобных сервисов с помощью Spring нужно уметь вызывать конечные точки REST других сервисов (рис. 11.1).

В этой главе вы познакомитесь со следующими тремя способами вызова конечных точек REST из Spring-приложения.

1. **OpenFeign** — инструмент из проекта Spring Cloud. Советую разработчикам применять его для вызова конечных точек REST в новых приложениях.
2. **RestTemplate** — инструмент, который применяется для вызова конечных точек REST, начиная со Spring 3. RestTemplate и сегодня популярен у разработчиков и широко используется в Spring-приложениях. Но, как вы скоро

узнаете, для него есть лучшая альтернатива — `OpenFeign`, поэтому в работе над новыми продуктами, пожалуй, стоит предпочесть именно ее.

3. `WebClient` — функция Spring, призванная стать заменой для `RestTemplate`. Основана на другой концепции — *реактивном программировании*, о котором мы поговорим в конце главы.

Конечные точки REST позволяют организовать обмен данными между двумя приложениями: одно приложение предоставляет доступ к своему функционалу другому приложению через HTTP



REST

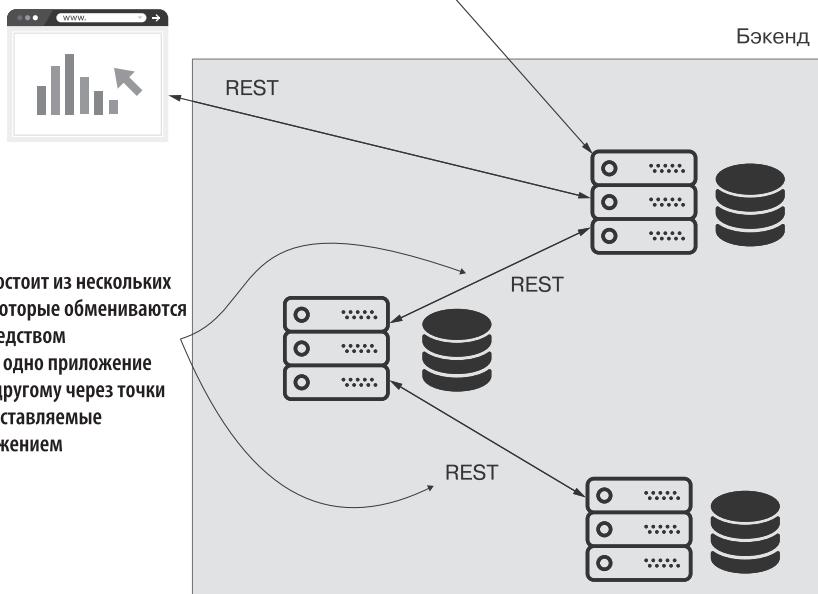


Рис. 11.1. Одно приложение бэкенда часто играет роль клиента для другого приложения бэкенда. Чтобы получить доступ к определенным данным, первое обращается к точкам доступа REST второго

Первым инструментом Spring, с которым мы познакомимся в разделе 11.1, будет `OpenFeign`, входящий в семейство Spring Cloud. В настоящее время я рекомендую использовать этот инструмент для всех новых проектов. Как вы узнаете, `OpenFeign` отличается несложным синтаксисом, который сильно упрощает вызов конечных точек REST из приложений Spring.

В разделе 11.2 мы будем использовать `RestTemplate`. Но предупреждаю: начиная с версии Spring 5, этот инструмент переведен в режим обслуживания, поэтому однажды устареет. Почему же тогда мы его изучаем? Дело в том, что `RestTemplate`

используется для вызова конечных точек REST в большинстве существующих сегодня проектов Spring — когда эти проекты зарождались, данный инструмент был единственным или наилучшим вариантом для реализации нужного функционала. Для некоторых из них возможностей `RestTemplate` вполне достаточно и сейчас, поэтому замена не имеет смысла. В других случаях внедрение нового решения потребовало бы слишком много времени и затраты бы не окупились. В любом случае Spring-разработчик все еще обязан уметь взаимодействовать с этим инструментом.

Есть один интересный факт, который обычно сбивает студентов с толку: в документации `RestTemplate` (<http://mng.bz/7lWe>) рекомендуется заменять `RestTemplate` на `WebClient`. В разделе 11.3 я объясню, почему этот совет не всегда работает. Мы обсудим `WebClient` и поймем, когда его следует использовать.

Чтобы изучить эти три фундаментальные методики, мы напишем примеры для них. Вначале разработаем проект, в котором используется конечная точка. Нашей целью будет вызов конечной точки каждым из трех описанных в главе способов: `OpenFeign`, `RestTemplate` и `WebClient`. Предположим, вы хотите создать приложение, позволяющее пользователям выполнять электронные платежи. Чтобы совершить платеж, нужно вызвать конечную точку другого приложения. Этот сценарий наглядно представлен на рис. 11.2, а на рис. 11.3 подробно показана схема передачи запроса и ответа.

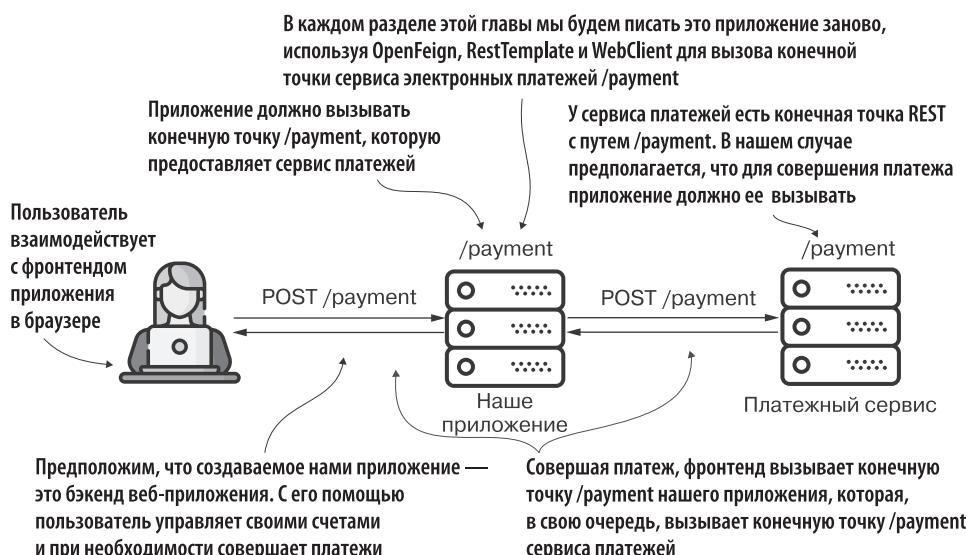


Рис. 11.2. Чтобы вы научились правильно вызывать конечные точки REST, мы рассмотрим несколько примеров. Для каждого создадим два проекта: один будет предоставлять конечную точку REST, а второй — демонстрировать вызов конечной точки REST посредством OpenFeign, RestTemplate и WebClient

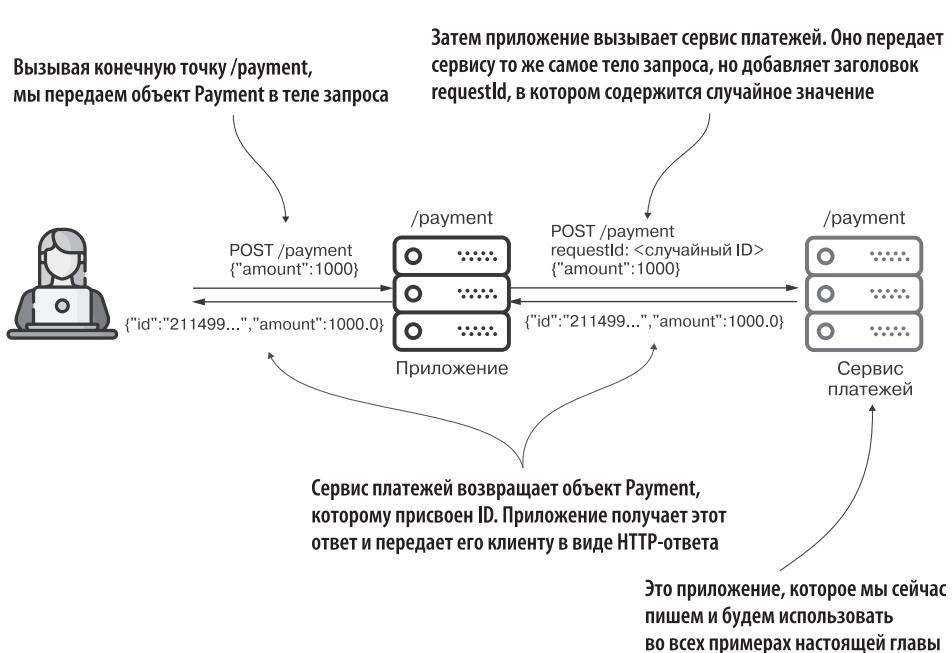


Рис. 11.3. Сервис платежей предоставляет конечную точку, которая принимает HTTP-запросы с телом запроса. Приложение отправляет запросы в конечную точку, используя OpenFeign, RestTemplate или WebClient

В первом проекте мы создадим приложение сервиса платежей — оно нам понадобится во всех остальных примерах.

Новый проект назовем `sq-ch11-payments`. Поскольку это веб-приложение, подобно другим проектам, описанным в главах 7–10, здесь нужно добавить веб-зависимость в файл `pom.xml`:

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

Модель платежа опишем в виде класса `Payment`:

```
public class Payment {
    private String id;
    private double amount;

    // Геттеры и сеттеры
}
```

Реализация конечной точки в классе контроллера показана в листинге 11.1. В техническом отношении здесь нет ничего сложного. Метод получает экземпляр

`Payment`, присваивает платежу случайный ID и возвращает его. Конечная точка очень проста, но вполне подходит для демонстрационных целей. В ней используется HTTP-метод POST. Нам нужно определить заголовок и тело запроса. При вызове конечной точки возвращается HTTP-ответ с заголовком и объектом `Payment` в теле запроса.

Листинг 11.1. Реализация конечной точки /payment

в классе контроллера

```
@RestController
public class PaymentsController {
    private static Logger logger = ← Используем вывод в консоль и убеждаемся, что
        Logger.getLogger(PaymentsController.class.getName()); ← при вызове конечной точки соответствующий метод
                                                                контроллера получает правильные данные

    @PostMapping("/payment") ← Приложение предоставляет конечную точку
    public ResponseEntity<Payment> createPayment( ← с HTTP-методом POST и путем /payment
        @RequestHeader String requestId, ← Конечная точка получает заголовок и тело
        @RequestBody Payment payment ← запроса от вызывающего ее приложения.
    ) { ← Эти данные передаются в метод
        logger.info("Received request with ID " + requestId + ← контроллера в качестве параметров
            " ;Payment Amount: " + payment.getAmount());

        payment.setId(UUID.randomUUID().toString()); ← Метод присваивает платежу
        return ResponseEntity ← случайный ID
            .status(HttpStatus.OK)
            .header("requestId", requestId)
            .body(payment);
    }
}
```

Если теперь выполнить приложение, оно запустит Tomcat на порте 8080 — как уже говорилось в главе 7, этот порт Spring Boot выбирает по умолчанию. Конечная точка будет доступна, и ее можно будет вызвать через cURL или Postman. Но цель данной главы — научиться создавать приложения, вызывающие конечные точки. Именно этим мы займемся в разделах 11.1, 11.2 и 11.3.

11.1. ВЫЗОВ КОНЕЧНОЙ ТОЧКИ REST С ПОМОЩЬЮ OPENFEIGN ИЗ SPRING CLOUD

Рассмотрим современную технологию вызова конечных точек REST. В большинстве Spring-приложений разработчики используют инструмент `RestTemplate` (о котором мы поговорим в разделе 11.2). Как мы помним, начиная с версии Spring 5, данный инструмент переведен в режим обслуживания. Более того, он скоро устареет, так что лучше начнем с инструмента, который я рекомендую использовать как альтернативу `RestTemplate`, — `OpenFeign`.

Как вы узнаете из примера ниже, для OpenFeign нужно написать только интерфейс — остальное этот инструмент сделает сам.

Чтобы понять, как функционирует OpenFeign, мы создадим проект sq-ch11-ex1 и разработаем приложение, в котором будем использовать OpenFeign для вызова конечной точки приложения sq-ch11-payments (рис. 11.4).

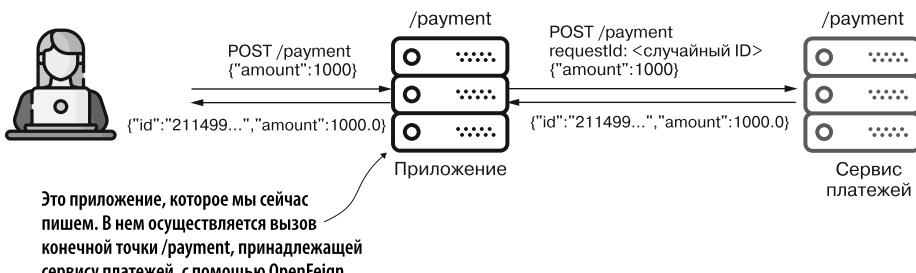


Рис. 11.4. Сейчас мы создадим приложение, которое обращается к конечной точке /payment, предоставляемой сервисом платежей. Нужную функциональность обеспечит OpenFeign

Мы определим интерфейс и объявим в нем методы, которые будут использовать конечные точки REST. Единственное, что для этого нужно, — снабдить методы аннотациями, где определены путь, HTTP-метод и при необходимости параметры, заголовки и тело запроса. Интересный момент: нам не нужно описывать сами методы. Как только мы определим те из них, которые будут основаны на аннотациях, фреймворк сам напишет их реализацию. Таким образом, мы снова полагаемся на прекрасную магию Spring.

На рис. 11.5 показана структура класса для создаваемого нами приложения, которое использует конечные точки REST.

В файле pom.xml нужно определить следующую зависимость:

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

Определив эту зависимость, мы можем создать прокси-интерфейс (как показано на рис. 11.5). В терминологии OpenFeign такой интерфейс также называют клиентом OpenFeign. OpenFeign создает его реализацию, так что нам не придется писать код,зывающий конечную точку, — хватит нескольких аннотаций, сообщающих, как именно нужно отправлять запрос. В листинге 11.2 видно, как легко создаются описания запросов при использовании OpenFeign.

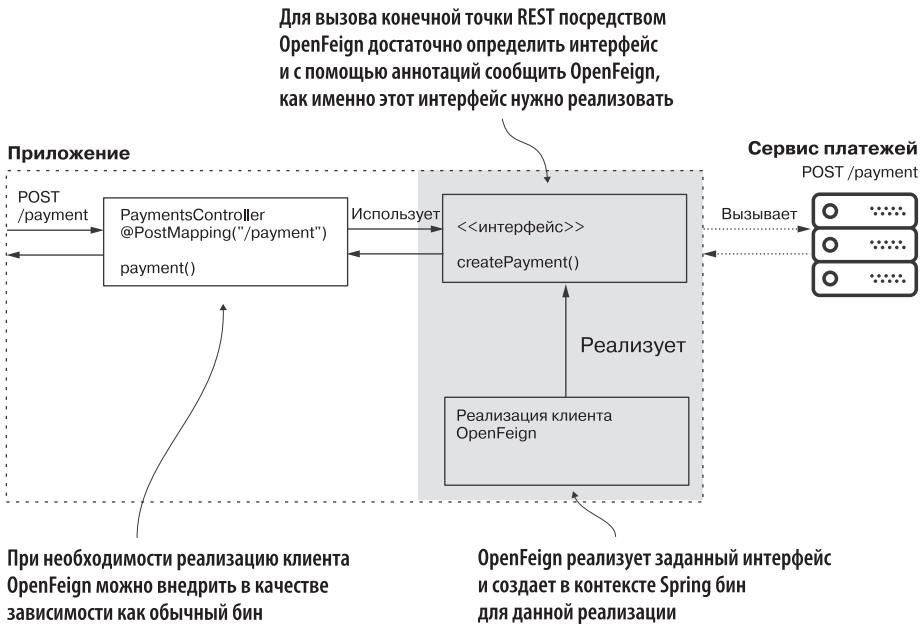


Рис. 11.5. Для OpenFeign достаточно определить интерфейс (контракт) и сообщить, где этот контракт находится. В соответствии с конфигурацией, описанной аннотациями, OpenFeign создаст реализацию интерфейса в виде бина, помещенного в контекст Spring. Этот бин затем можно будет внедрить в любой объект приложения

Листинг 11.2. Объявление клиентского интерфейса OpenFeign

```
@FeignClient(name = "payments", url = "${name.service.url}")
public interface PaymentsProxy {
    @PostMapping("/payment")
    Payment createPayment(
        @RequestHeader String requestId,
        @RequestBody Payment payment);
}
```

Создаем REST-клиента с помощью аннотации `@FeignClient`. Минимальная конфигурация подразумевает имя и базовый URI конечной точки

Определяем путь и HTTP-метод конечной точки

Определяем заголовки и тело HTTP-запроса

Прежде всего необходимо снабдить интерфейс аннотацией `@FeignClient`. Она сообщает OpenFeign, что нужно создать реализацию данного контракта. С помощью атрибута `name` аннотации `@FeignClient` определяется имя прокси для внутреннего использования в OpenFeign. Это имя однозначно идентифицирует данного клиента в приложении. В `@FeignClient` также формируется базовый URI запроса, представляющий собой строку, которая является значением атрибута `url`.

ПРИМЕЧАНИЕ

Всегда храните URI и другие параметры, которые могут зависеть от среды, только в файлах свойств. Никогда не вписывайте их жестко в код приложения.

Свойства проекта хранятся в файле `application.properties` и доступны в коде с помощью синтаксиса `${property_name}`. Благодаря этому вам не придется заново компилировать код всякий раз, когда потребуется запускать приложение в новой среде.

Каждый метод, объявленный в интерфейсе, соответствует вызову конечной точки REST. Для предоставления доступа к точкам для действий контроллера применяются те же аннотации, которые вам уже знакомы из главы 10:

- для определения пути и HTTP-метода: `@GetMapping`, `@PostMapping`, `@PutMapping` и т. п.;
- для определения заголовка запроса: `@RequestHeader`;
- для определения тела запроса: `@RequestBody`.

Возможность многократного использования аннотаций (одних и тех же и для `OpenFeign`, и для определения конечных точек) мне кажется очень полезной. Получается, для работы с `OpenFeign` вам не придется специально что-то изучать — просто используйте те же инструменты, что и в `Spring MVC`.

Нам нужно сообщить `OpenFeign`, где находятся интерфейсы, определяющие контракты клиентов. Для этого воспользуйтесь аннотацией `@EnableFeignClients`. В листинге 11.3 представлен класс конфигурации проекта, в котором активированы клиенты `OpenFeign`.

Листинг 11.3. Активация клиентов OpenFeign в классе конфигурации

```
@Configuration
@EnableFeignClients(←
    basePackages = "com.example.proxy")
public class ProjectConfig { }
```

Активируем клиентов OpenFeign и сообщаем
зависимости OpenFeign, где следует искать
прокси-контракты

Теперь можно внедрить клиент `OpenFeign` через интерфейс, определенный в листинге 11.2. Поскольку инструмент `OpenFeign` активирован, он знает, как реализовать интерфейсы с аннотациями `@FeignClient`. Как уже говорилось в главе 5, `Spring` сам знает, как создать экземпляр бина из контекста при использовании абстракций, — именно это мы сейчас и делаем. В листинге 11.4 представлен класс контроллера, в котором внедряется `OpenFeign`.

Листинг 11.4. Внедрение и использование клиента OpenFeign

```

@RestController
public class PaymentsController {

    private final PaymentsProxy paymentsProxy;

    public PaymentsController(PaymentsProxy paymentsProxy) {
        this.paymentsProxy = paymentsProxy;
    }

    @PostMapping("/payment")
    public Payment createPayment(
        @RequestBody Payment payment
    ) {
        String requestId = UUID.randomUUID().toString();
        return paymentsProxy.createPayment(requestId, payment);
    }
}

```

Теперь запустим оба проекта (сервис платежей и приложение, которое мы написали в этом разделе) и вызовем конечную точку `/payment` приложения, используя cURL или Postman. В случае с cURL команда запроса выглядит так:

```

curl -X POST -H 'content-type:application/json' -d '{"amount":1000}'
➥ http://localhost:9090/payment

```

После выполнения команды cURL в консоли появится такой ответ:

```
{"id":"1c518ead-2477-410f-82f3-54533b4058ff","amount":1000.0}
```

В консоли сервиса платежей вы обнаружите запись, подтверждающую, что приложение корректно передало запрос сервису платежей:

```

Received request with ID 1c518ead-2477-410f-82f3-54533b4058ff ;Payment
➥ Amount: 1000.0

```

11.2. ВЫЗОВ КОНЕЧНЫХ ТОЧЕК REST С ПОМОЩЬЮ RESTTEMPLATE

Далее мы снова создадим приложение,зывающее конечную точку `/payment` сервиса платежей, но на этот раз воспользуемся другим инструментом — `RestTemplate`.

Мне бы не хотелось, чтобы вы думали, будто с `RestTemplate` что-то не так. Этот инструмент отправляют на покой не потому, что он работает неправильно,

и не потому, что он плохой. Просто приложения совершенствуются и требуют более широкого функционала. Разработчики хотят пользоваться разными технологиями, но не ко всем из них можно легко получить доступ при помощи `RestTemplate`. Среди них:

- синхронный и асинхронный вызов конечных точек;
- возможность писать меньше кода и обрабатывать меньше исключений (избегать шаблонного кода);
- возможность выполнять повторные вызовы и операции отката (логика, которая осуществляется, когда приложение по какой-то причине не может выполнить определенный вызов REST).

Другими словами, разработчики предпочитают по возможности получать больше готовых функций, чтобы меньше писать код самим. Напомню: как раз в этом — в повторном использовании кода и возможности избежать шаблонного кода — заключается главная цель применения фреймворков. Сравнив примеры, созданные нами в разделах 11.1 и 11.2, вы сможете оценить сами, насколько проще использовать `OpenFeign` по сравнению с `RestTemplate`.

ПРИМЕЧАНИЕ

Я вынес из своего опыта важный урок: когда что-то объявляют устаревшим, еще не значит, что это не стоит изучить. Иногда подобные технологии продолжают использоваться в проектах на протяжении многих лет, после того как их признали неактуальными. В частности, к ним относятся `RestTemplate` и проект `Spring Security OAuth`.

Чтобы создать вызов, нужно выполнить следующие операции (рис. 11.6).

1. Описать HTTP-заголовки, создав и настроив конфигурацию экземпляра `HttpHeaders`.
2. Создать экземпляр `HttpEntity`, в котором представлены данные запроса (заголовки и тело).
3. С помощью метода `exchange()` отправить HTTP-вызов и получить HTTP-ответ.

Выполним этот пример. Для начала создадим проект `sq-ch11-ex2`. Определение прокси-класса вы найдете в листинге 11.5. Обратите внимание, как описан заголовок с помощью экземпляра `HttpHeaders` в методе `createPayment()` и как с помощью метода `add()` к этому экземпляру добавлен заголовок `requestId`. На базе имеющихся заголовков и тела (которые метод получает в качестве параметров) создается экземпляр `HttpEntity`. После этого с помощью принадлежащего `RestTemplate` метода `exchange()` отправляется HTTP-запрос. Параметрами `exchange()` являются URI и HTTP-метод, а также экземпляр `HttpEntity` (в котором хранятся данные запроса) и тип, который ожидается в теле ответа.

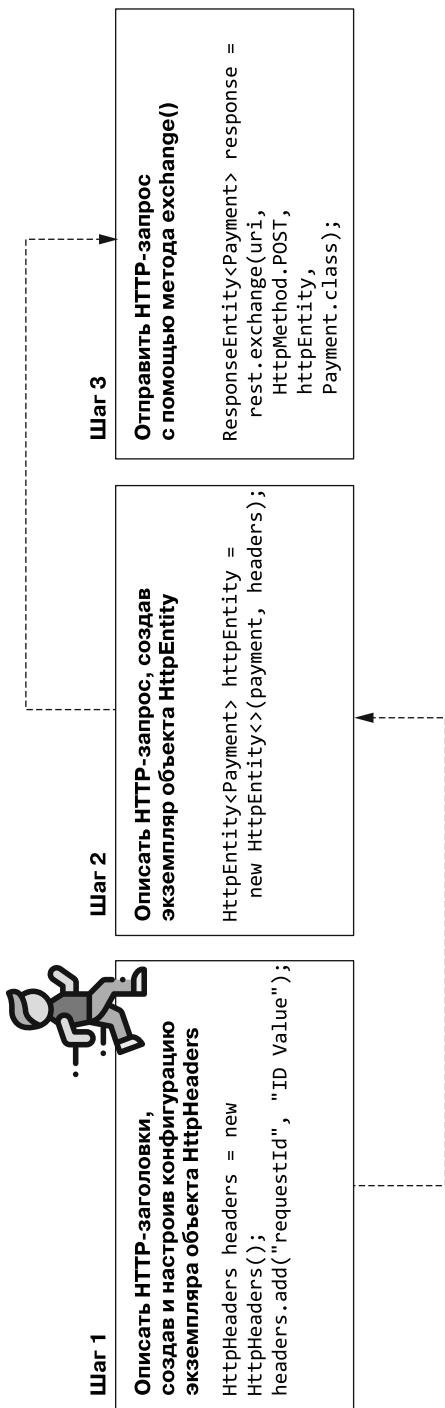


Рис. 11.6. Чтобы создать более сложный HTTP-запрос, нужно описать заголовки с помощью класса HttpHeaders, а затем применить класс HttpEntity для определения данных запроса. Как только эти данные будут сформированы, нужно отправить запрос с помощью метода exchange()

Листинг 11.5. Класс PaymentsProxy, вызывающий конечную точку /payment

```

@Component
public class PaymentsProxy {

    private final RestTemplate rest;

    @Value("${name.service.url}")
    private String paymentsServiceUrl; ← Получаем из файла свойств
                                                URL сервиса платежей

    public PaymentsProxy(RestTemplate rest) { ← С помощью DI внедряем
        this.rest = rest;
    }                                         в конструкторе RestTemplate
                                                из контекста Spring

    public Payment createPayment(Payment payment) {
        String uri = paymentsServiceUrl + "/payment";

        HttpHeaders headers = new HttpHeaders();
        headers.add("requestId",
                    UUID.randomUUID().toString()); ← Создаем объект HttpHeaders,
                                                в котором определяются
                                                заголовки HTTP-запроса

        HttpEntity<Payment> httpEntity = ← Строим объект HttpEntity
            new HttpEntity<>(payment, headers); ← с данными запроса

        ResponseEntity<Payment> response =
            rest.exchange(uri,
                           HttpMethod.POST,
                           httpEntity,
                           Payment.class); ← Отправляем HTTP-запрос
                                                и извлекаем данные из HTTP-ответа

        return response.getBody(); ← Возвращаем тело HTTP-запроса
    }
}

```

Мы определили простую конечную точку, которую можно будет вызывать с помощью этой реализации — такую же, как в разделе 11.1. В листинге 11.6 показано определение класса контроллера.

Листинг 11.6. Класс контроллера для тестирования реализации

```

@RestController
public class PaymentsController {

    private final PaymentsProxy paymentsProxy;

    public PaymentsController(PaymentsProxy paymentsProxy) {
        this.paymentsProxy = paymentsProxy;
    } ← Создаем действие контроллера
        и связываем его с путем /payment

    @PostMapping("/payment") ← Извлекаем данные о платеже из тела запроса
    public Payment createPayment(
        @RequestBody Payment payment ←
        ) {
        return paymentsProxy.createPayment(payment); ←
    }                                         Вызываем прокси-метод, который, в свою очередь, вызывает конечную
                                                точку сервиса платежа. Получаем тело запроса и возвращаем его клиенту
}

```

Запустив оба приложения — сервис платежей (проект sq-ch11-payments) и новое приложение (проект sq-ch11-ex2) — на разных портах, мы убедимся, что все работает, как предполагалось. В данном случае я оставил неизменной часть конфигурации из раздела 11.1, использовав порт 8080 для сервиса платежей и порт 9090 для нового приложения.

Чтобы вызвать конечную точку приложения, можно воспользоваться следующей командой с URL:

```
curl -X POST -H 'content-type:application/json' -d '{"amount":1000}'  
→ http://localhost:9090/payment
```

В результате в консоли появится следующий ответ:

```
{  
    "id":"21149959-d93d-41a4-a0a3-426c6fd8f9e9",  
    "amount":1000.0  
}
```

В консоли сервиса платежей вы найдете журнал, подтверждающий, что приложение правильно отправило запрос сервиса:

```
Received request with ID e02b5c7a-c683-4a77-bd0e-38fe76c145cf ;Payment  
→ Amount: 1000.0
```

11.3. ВЫЗОВ КОНЕЧНОЙ ТОЧКИ REST С ПОМОЩЬЮ WEBCLIENT

Обсудим вызов конечных точек REST с помощью `WebClient`. `WebClient` — это инструмент, используемый в различных приложениях и построенный на базе методологии, называемой *реактивным программированием*. Реактивное программирование — продвинутый подход, и я рекомендую приступить к его изучению только после того, как вы хорошо усвоите основы. Начните с книги Craig Walls, *Spring in Action*, 6th ed. (Manning, 2021).

В документации Spring рекомендуется использовать `WebClient`, но это имеет смысл только для приложений, написанных по технологии реактивного программирования. Если вы не создаете реактивный продукт, лучше применять `OpenFeign`. Как и все остальное в мире программного обеспечения, `WebClient` хорошо работает в одних случаях и может создавать проблемы в других. Выбирая `WebClient` для вызова конечных точек REST, вы обрекаете приложение на то, чтобы быть реактивным.

ПРИМЕЧАНИЕ

Если вы решили не делать приложение реактивным, реализуйте клиентские функции REST с помощью `OpenFeign`. Если вы создаете реактивное приложение, необходимо использовать соответствующий реактивный инструмент — `WebClient`.

Несмотря на то что реактивные приложения немного выходят за рамки понятия «основы программирования», я бы все же хотел, чтобы вы имели представление об использовании `WebClient`, знали, чем этот инструмент отличается от остальных, рассмотренных выше, и могли сравнить разные подходы к программированию. Я немного расскажу о реактивных приложениях, после чего мы поработаем с `WebClient`, чтобы вызвать конечную точку `/payment`, которую мы использовали в качестве примера в разделах 11.1 и 11.2.

В нереактивных приложениях процесс бизнес-логики выполняется в потоке. Этот процесс состоит из множества задач, которые не являются независимыми друг от друга. Все они выполняются в одном потоке. Рассмотрим пример, чтобы проследить, в каких случаях при таком подходе могут возникнуть проблемы и что здесь можно улучшить.

Предположим, мы разрабатываем банковское приложение, в котором у пользователя есть один или несколько кредитных счетов. Мы создаем компонент, который вычисляет общий долг клиента банка. Этот функционал реализуют другие элементы системы, выполняя REST-вызовы, где передается уникальный ID пользователя. Для вычисления долга в разрабатываемом нами процессе проводятся следующие операции (рис. 11.7).

1. Приложение получает ID пользователя.
2. Приложение вызывает другой сервис системы, который определяет, есть ли у данного пользователя кредиты в других учреждениях.
3. Приложение вызывает еще один сервис системы, сообщающий о долгах пользователя по внутренним кредитам.
4. Если у пользователя есть внешние долги, приложение вызывает внешний сервис, который вычисляет размер внешнего долга.
5. Приложение суммирует все долги и возвращает значение в виде HTTP-ответа.

Это всего лишь примерные шаги функционала, но я составил схему, чтобы показать, в каких случаях может быть полезно создание реактивного приложения. Тщательно проанализируем эти операции. На рис. 11.8 показана схема выполнения рассматриваемого сценария с точки зрения потока.

Для каждого запроса приложение создает новый поток, где операции выполняются последовательно, одна за другой. Прежде чем перейти к следующей операции, поток вынужден ожидать, пока не завершится предыдущая. Всякий раз, когда приложение выполняет операцию ввода-вывода, поток блокируется.

Создаваемый нами функционал сначала должен вызвать другой сервис системы, чтобы получить данные о пользователе. Пока выполняется это действие, приложение ждет ответа, после чего может перейти к п. 3. Любой запрос к другому компоненту — это операция ввода-вывода, которая занимает какое-то время. Пока выполняется вызов, поток заблокирован и не может делать что-либо еще

Другое приложение системы делает вызов. Оно передает ID пользователя, чтобы вычислить общий долг клиента

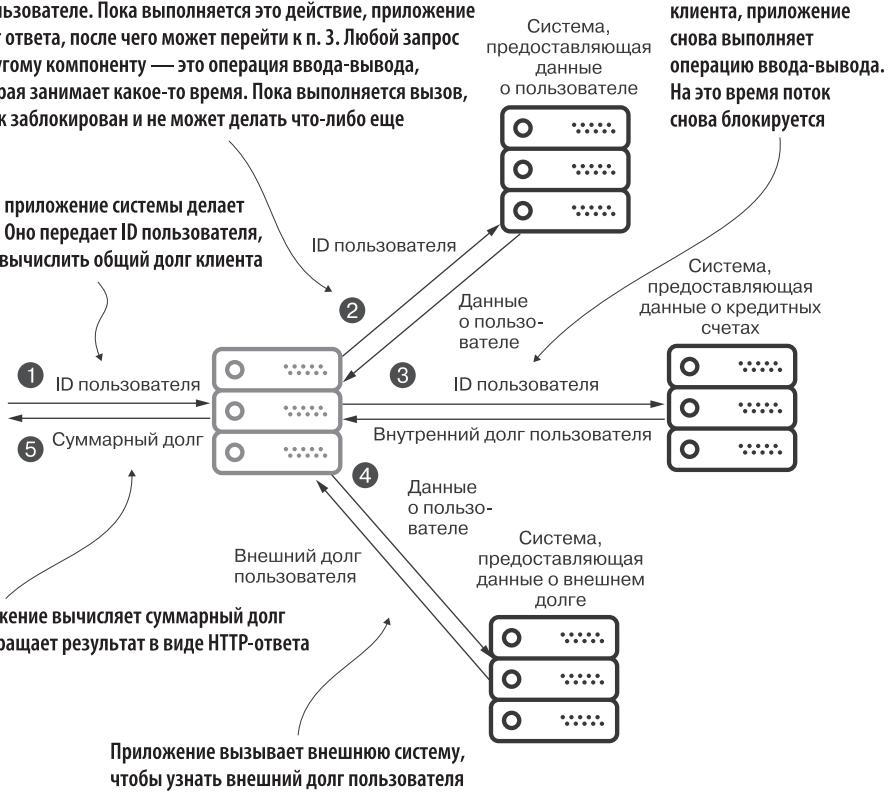


Рис. 11.7. Схема функционала, демонстрирующая полезность реактивного программирования. Чтобы вычислить суммарный долг пользователя, банковское приложение вызывает несколько других приложений. Из-за этих вызовов поток, выполняющий запрос, несколько раз блокируется, ожидая завершения операций ввода-вывода

Здесь мы наблюдаем две серьезные проблемы.

1. Операции ввода-вывода блокируют поток: пока они выполняются, поток бездействует и лишь занимает место в памяти приложения. Он потребляет ресурсы, не принося никакой пользы. При таком подходе возможны ситуации, когда приложение получит сразу десять запросов, но все потоки будут заняты, ожидая информацию от других систем.
2. Некоторые задачи не зависят друг от друга. Например, приложение может одновременно выполнять п. 2 и 3. Нет необходимости ждать, пока закончится

действие в п. 2, чтобы перейти к п. 3. Приложению просто нужно в итоге получить результат обеих операций и вычислить суммарный долг.

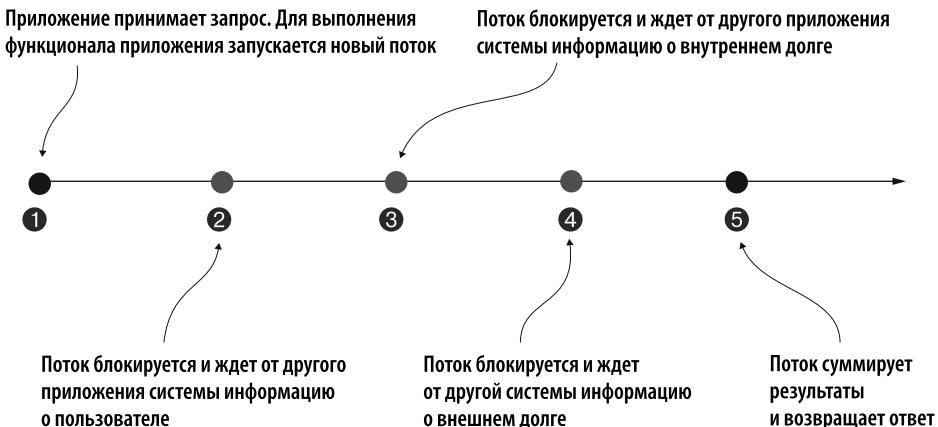


Рис. 11.8. Реализация функционала приложения с точки зрения потока. Стрелка изображает последовательность работы. Некоторые операции по получению информации приводят к блокированию потока: поток ждет завершения этих задач и лишь после этого продолжает выполнение

При реактивном программировании мы отказываемся от идеи существования одного неделимого процесса, все задачи которого от начала и до конца выполняются в единственном потоке. В реактивных приложениях задачи считаются независимыми друг от друга, и для выполнения процесса, состоящего из нескольких задач, могут совместно использоваться несколько потоков.

Теперь функционал представляется не как последовательность операций, расположенных вдоль одной временной оси, а как пул задач, которые выполняет целая команда разработчиков. Пользуясь данной аналогией, вам будет проще представить работу реактивного приложения: разработчики — это потоки, а пул задач — операции функционала.

Два разработчика могут выполнять две разные задачи одновременно, если они не зависят одна от другой. Если задача стопорится из-за внешней зависимости, разработчик может временно ее приостановить и заняться чем-нибудь еще. Когда она разблокируется, разработчик сможет снова к ней вернуться или же эту задачу закончит его коллега (рис. 11.9).

При таком подходе нам больше не нужен отдельный поток для каждого запроса. Мы можем обрабатывать несколько запросов и с меньшим количеством потоков, так как им больше не приходится простоять. Если выполнение одной из задач блокируется, поток может оставить ее и заняться другой, свободной.

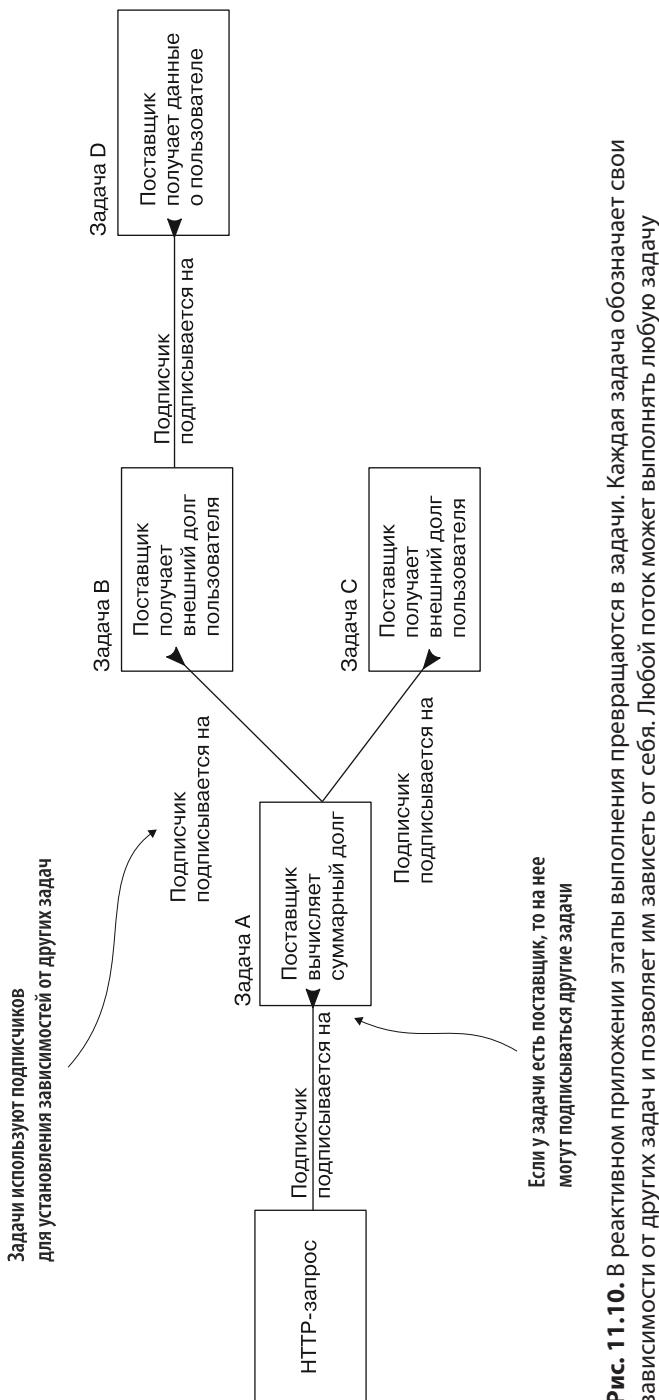
В техническом отношении в реактивном приложении создается процесс, в котором определены задачи и зависимости между ними. Спецификация реактивного приложения предусматривает два компонента для представления зависимостей между задачами: поставщик и подписчик.



Рис. 11.9. Реактивное приложение работает примерно так. Поток не берет задачи запроса по порядку и не бездействует, пока они заблокированы. Вместо этого все задачи всех запросов образуют единый пул. Любой свободный поток может выполнить любую задачу любого запроса. Таким образом, независимые задачи могут выполняться параллельно и потоки не бездействуют

Задача возвращает поставщика, который позволяет другим задачам подписываться на него, тем самым обозначая их зависимость. Задача использует подписчика, чтобы присоединиться к поставщику другой задачи и получить результат выполнения этой задачи, когда она завершится.

На рис. 11.10 показан обсуждавшийся нами ранее сценарий, реализованный по принципу реактивного программирования. Уделите пару минут и сравните эту схему с рис. 11.8. Теперь задачи не являются последовательными точками на шкале времени — они не зависят от потока и сами объявляют свои зависимости. Задачи выполняются несколькими потоками, и ни одному из них не приходится ждать разблокировки конкретной задачи, пока проводится операция ввода-вывода. Вместо этого поток может выполнить другую задачу.



Более того, задачи, зависящие одна от другой, также могут выполняться одновременно. На рис. 11.10 операции С и D, соответствующие п. 2 и 3 исходного процесса, могут проводиться параллельно, что повышает производительность приложения.

В примере нам понадобятся проекты sq-ch11-payments (сервис платежей) и sq-ch11-ex3 (приложение). Мы использовали сервис платежей в разделах 11.1 и 11.2, где он предоставлял доступ к конечной точке `/payment` посредством HTTP-метода POST. Сейчас же мы будем отправлять запросы к конечной точке сервиса платежей посредством инструмента `WebClient`.

Поскольку `WebClient` предполагает реактивный подход, вместо стандартной веб-зависимости нужно добавить зависимость, которая называется `WebFlux`. В следующем фрагменте кода показано, как добавить `WebFlux` в файл `pom.xml` готового проекта или при создании проекта с помощью `start.spring.io`:

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

Для вызова конечной точки REST нам понадобится экземпляр `WebClient`. Наилучший способ организовать удобный доступ к такому экземпляру — поместить бин в контекст Spring, создав в классе конфигурации метод с аннотацией `@Bean`, как было показано в главе 2. Класс конфигурации приложения представлен в листинге 11.7.

Листинг 11.7. Добавление бина `WebClient` в контекст Spring в классе конфигурации

```
@Configuration
public class ProjectConfig {

    @Bean
    public WebClient webClient() {
        return WebClient
            .builder() ←———— Создаем бин WebClient и добавляем его в контекст Spring
            .build();
    }
}
```

В листинге 11.8 показан прокси-класс, где используется `WebClient` для вызова конечной точки, предоставляемой приложением. Логика класса аналогична той, с которой вы познакомились в разделе о `RestTemplate`: извлекаем из файла свойств базовый URL, определяем HTTP-метод, заголовки и тело, после чего осуществляем вызов. Методы `WebClient` называются иначе, но по их именам нетрудно понять, что именно они делают.

Листинг 11.8. Прокси-класс с использованием WebClient

```

@Component
public class PaymentsProxy {

    private final WebClient webClient;

    @Value("${name.service.url}")
    private String url;                                | Извлекаем базовый URL  
из файла свойств

    public PaymentsProxy(WebClient webClient) {
        this.webClient = webClient;
    }

    public Mono<Payment> createPayment(
        String requestId,
        Payment payment) {
        return webClient.post()                         | Определяем HTTP-метод, который  
будет использоваться при вызове
            .uri(url + "/payment")                     | Определяем URI вызова
            .header("requestId", requestId)           | Добавляем к запросу HTTP-заголовок.  
Для создания нескольких заголовков  
можно вызывать метод header()  
несколько раз
            .body(Mono.just(payment), Payment.class)  | Формируем тело HTTP-запроса
            .retrieve()                                | Отправляем HTTP-запрос  
и получаем HTTP-ответ
            .bodyToMono(Payment.class);                | Извлекаем тело HTTP-запроса
    }
}

```

В нашем примере мы создали класс с именем `Mono`. Этот класс определяет поставщика. Вы найдете его в листинге 11.8, где метод, выполняющий вызов, получает входные данные не непосредственно, а через `Mono`. Благодаря этому можно создать независимую задачу, которая будет формировать тело запроса. `WebClient` является подписчиком этой задачи и, таким образом, зависит от нее.

Данный метод также не возвращает значение непосредственно. Вместо этого он возвращает экземпляр `Mono`, что позволяет другим функциям подписываться на него. Таким образом, процесс в приложении формируется не путем соединения задач в поток, а через установление зависимостей между ними посредством поставщиков и потребителей (рис. 11.11).

В листинге 11.8 также показан прокси-метод, который принимает экземпляр `Mono`, формируемый из тела HTTP-запроса, и возвращает этот экземпляр задаче, на которую подписан функционал `WebFlux`.

Дабы убедиться, что все работает правильно, мы, как и в предыдущих примерах этой главы, создадим класс контроллера, использующий прокси для предоставления доступа к конечной точке. Мы будем вызывать эту конечную точку, чтобы протестировать поведение нашего приложения. Класс контроллера представлен в листинге 11.9.

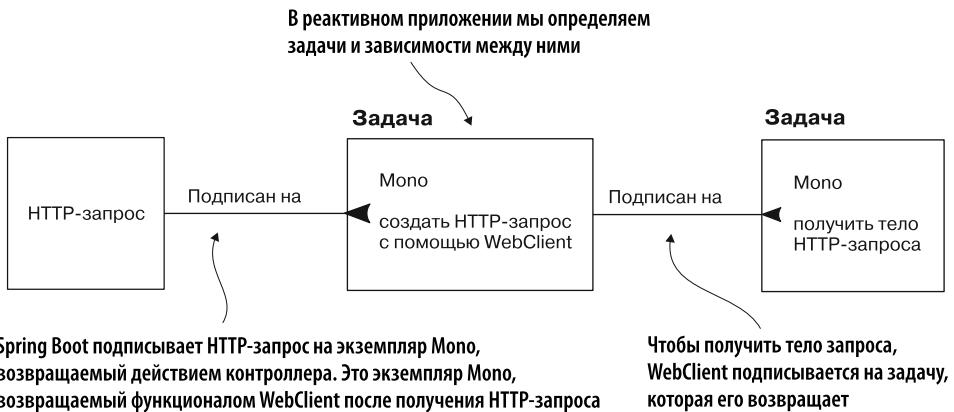


Рис. 11.11. Цепочка задач в реактивном приложении. Разрабатывая реактивное веб-приложение, мы создаем задачи и определяем зависимости между ними. Функционал WebFlux подписывает HTTP-запрос на созданную нами задачу посредством поставщика, который возвращается действием контроллера. В нашем случае мы получаем поставщика посылая HTTP-запрос с помощью WebClient. Чтобы WebClient отправил запрос, он подписывается на другую задачу, которая создает тело запроса

Листинг 11.9. Класс контроллера, предоставляющий доступ к конечной точке и вызывающий прокси

```

@RestController
public class PaymentsController {

    private final PaymentsProxy paymentsProxy;

    public PaymentsController(PaymentsProxy paymentsProxy) {
        this.paymentsProxy = paymentsProxy;
    }

    @PostMapping("/payment")
    public Mono<Payment> createPayment(
        @RequestBody Payment payment
    ) {
        String requestId = UUID.randomUUID().toString();
        return paymentsProxy.createPayment(requestId, payment);
    }
}

```

Чтобы протестировать работу обоих приложений — проекта sq-ch11-payments (сервис платежей) и проекта sq-ch11-ex3 — можно вызвать конечную точку /payment с помощью cURL или Postman. В случае cURL команда запроса выглядит так:

```

curl -X POST -H 'content-type:application/json' -d '{"amount":1000}'
→ http://localhost:9090/payment

```

После выполнения команды curl увидим в консоли примерно такой ответ:

```
{  
    "id": "e1e63bc1-ce9c-448e-b7b6-268940ea0fcc",  
    "amount": 1000.0  
}
```

В консоли сервиса платежей появится запись, подтверждающая, что приложение, разработанное в этом разделе, правильно отправляет запросы сервису платежей:

```
Received request with ID e1e63bc1-ce9c-448e-b7b6-268940ea0fcc ;Payment  
↳ Amount: 1000.0
```

РЕЗЮМЕ

- В реальных решениях вам часто будут встречаться сценарии использования, когда одно серверное приложение вызывает конечные точки, предоставляемые другим серверным приложением.
- В Spring есть несколько вариантов реализации клиентской части REST-сервисов. Из них наиболее актуальны три:
 - **OpenFeign** — решение, предложенное проектом Spring Cloud. Существенно упрощает код, который необходимо написать для вызова конечной точки REST. Имеет еще несколько функций, позволяющих создавать современные сервисы;
 - **RestTemplate** — простой инструмент, используемый для создания конечных точек REST в Spring-приложениях;
 - **WebClient** — реактивное решение для вызова конечных точек REST в Spring-приложениях.
- Не стоит применять **RestTemplate** при создании новых приложений. Для вызова конечных точек REST в этом случае лучше подойдет **OpenFeign** или **WebClient**.
- Если приложение построено по стандартной (нереактивной) технологии, стоит использовать **OpenFeign**.
- **WebClient** — отличный инструмент для приложений, созданных по принципу реактивного программирования. Но прежде, чем его применить, необходимо как следует изучить реактивное программирование и понять, как этот принцип реализуется в Spring.

19

Использование источников данных в Spring-приложениях

В этой главе

- ✓ Источники данных.
- ✓ Настройка конфигурации источника данных в Spring-приложении.
- ✓ Взаимодействие с базой данных с помощью `JdbcTemplate`.

Практически любое современное приложение должно где-то хранить информацию, с которой оно работает, и как-то ею управлять. Для этого обычно используются базы данных. Вот уже много лет реляционные базы данных обеспечивают приложениям простой и изящный способ хранения информации, который успешно применяется во многих сценариях использования. Spring-приложения, как и другие приложения, часто используют базы данных, поэтому вам необходимо научиться применять эти возможности в приложениях, написанных на основе Spring.

В этой главе мы обсудим, что такое источник данных, и рассмотрим наиболее простой способ организации доступа Spring-приложения к базе данных — с помощью инструмента `JdbcTemplate`, предоставляемого Spring.

На рис. 12.1 показано, какие фундаментальные свойства Spring вы уже научились применять в предыдущих главах. Вы проделали большой путь и теперь можете, используя Spring, разрабатывать функции для различных частей системы.

Теперь вы умеете создавать сервисы для обмена данными между Spring-приложениями и их клиентами (например, мобильными приложениями или веб-приложениями, запускаемыми через браузер)

Сейчас вы находитесь здесь! Вы научитесь организовывать хранение данных в Spring-приложениях

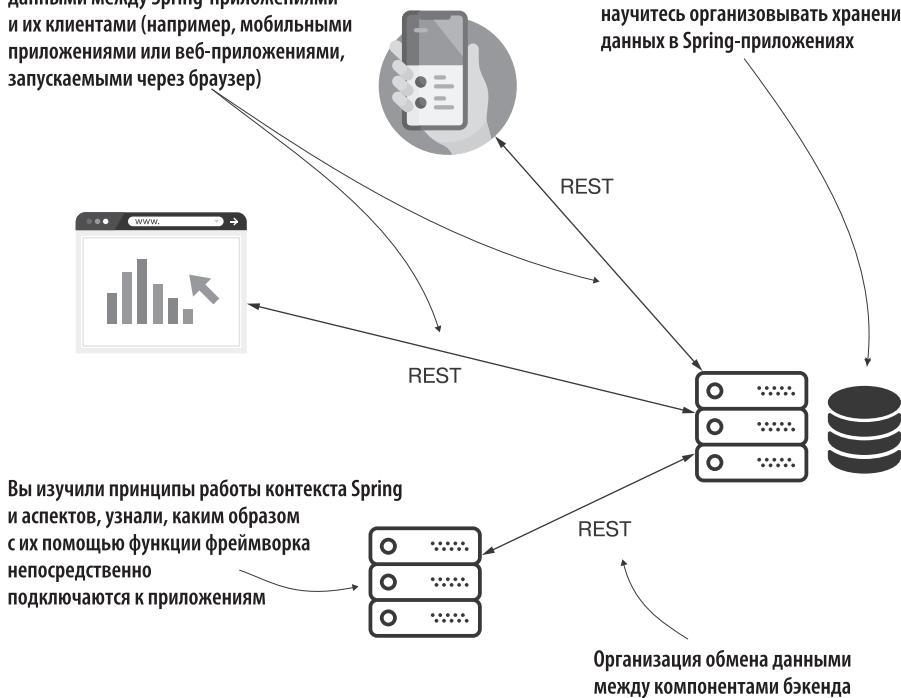


Рис. 12.1. Вы уже освоили основные части системы, реализуемые с помощью Spring. В главах 1–6 вы изучили базу — то, благодаря чему возможности Spring можно использовать при разработке приложений. В главах 7–11 вы научились создавать веб-приложения и применять конечные точки REST для обмена данными между компонентами системы. Теперь вы готовы приступить к получению важных навыков, позволяющих создавать Spring-приложения, которые взаимодействуют с хранилищами данных

12.1. ЧТО ТАКОЕ ИСТОЧНИК ДАННЫХ

Ниже мы рассмотрим важный элемент, необходимый Spring-приложению для доступа к базе данных: источник данных. Источник данных (рис. 12.2) — это компонент, отвечающий за соединение с сервером, обслуживающим базу данных (с системой управления базой данных — СУБД).

ПРИМЕЧАНИЕ

СУБД — это программное обеспечение, назначение которого — организовать эффективное управление имеющимися данными (добавление, изменение, получение) и сохранение их безопасности. СУБД управляет данными, хранящимися в базе данных. База данных — это постоянный набор данных.

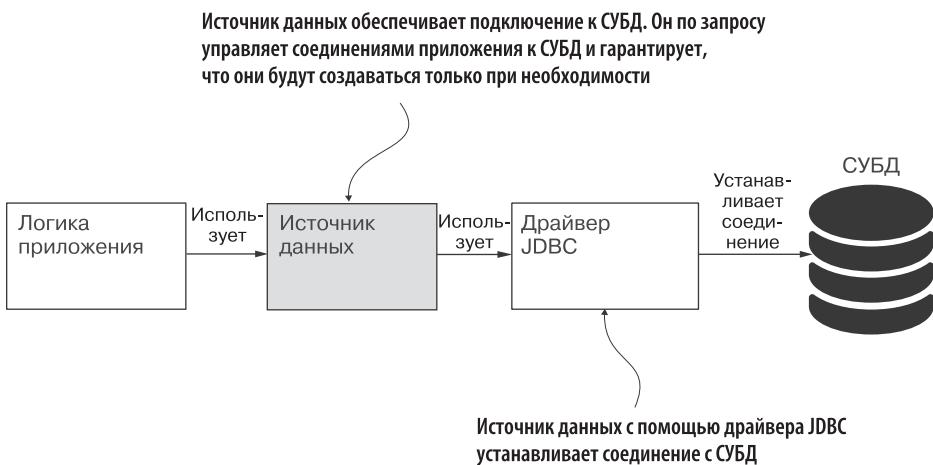


Рис. 12.2. Источник данных — это компонент, устанавливающий соединение с системой управления базой данных (СУБД). Для установки соединений, которыми он управляет, источник данных использует драйвер JDBC. Назначение источника данных — повысить производительность приложения, позволяя логике приложения повторно использовать соединения с СУБД и устанавливать новые соединения только при необходимости. Источник данных также обеспечивает закрытие соединений, когда они больше не нужны

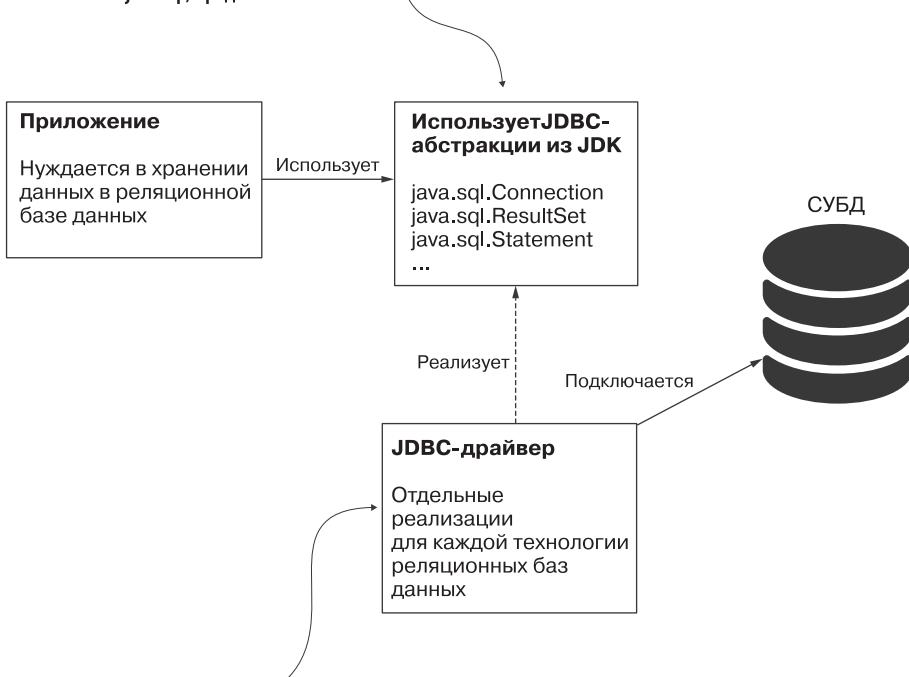
Если бы не существовало объекта, отвечающего за источник данных, приложению пришлось бы для каждой операции с данными устанавливать новое подключение к базе. На практике это нереально, поскольку подобный процесс кардинально замедлит приложение и приведет к проблемам с производительностью. Источник данных позволяет сделать так, чтобы приложение запрашивало установку нового соединения только тогда, когда это действительно необходимо, что значительно повышает его производительность.

Работа с любым инструментом, имеющим отношение к хранению данных в реляционной базе, в Spring подразумевает наличие источника данных. Поэтому нам важно сначала выяснить место источника данных на уровне управления данными приложения. Затем мы рассмотрим реализацию уровня хранения данных на примерах.

В приложениях Java функционал языка для соединения с реляционной базой данных называется Java Database Connectivity (JDBC). JDBC — это способ установить подключение к СУБД для взаимодействия с базой данных. Но в JDK нет отдельных реализаций для работы с конкретными технологиями (такими как MySQL, Postgres или Oracle). JDK только предоставляет абстракции для объектов, необходимых приложению для соединения с базой. Чтобы получить реализацию этой абстракции и позволить приложению подключаться к конкретной СУБД, необходимо добавить так называемый JDBC-драйвер — зависимость

реального времени (рис. 12.3). Каждый производитель дает свой драйвер, который следует подключить к приложению, чтобы оно могло соединяться с СУБД, построенной по соответствующей технологии. JDBC-драйверы не поставляются в комплекте с JDK или фреймворком, таким как Spring.

Приложение использует абстракции JDBC из JDK. В приложениях, которым для связи с базой данных нужны только JDBC, обычно применяются такие интерфейсы, как Connection, Statement и ResultSet из пакета java.sql, предоставляемого JDK



Но одних лишь абстракций недостаточно. Приложению нужны их реализации, позволяющие подключаться к конкретной базе данных. JDBC-драйверы обеспечивают такие реализации для разных типов СУБД. Например, если приложение должно подключаться к серверу баз данных MySQL, то нужно установить JDBC-драйвер MySQL, который реализует JDBC-абстракции, предоставляемые JDK, и определяет способ подключения к серверу

Рис. 12.3. Java-приложение устанавливает соединение с базой данных через СУБД. JDK предоставляет набор абстракций, но, кроме них, приложению также нужна их реализация для того типа реляционной базы данных, к которой подключается приложение. Эту реализацию предоставляет зависимость реального времени, называемая JDBC-драйвером

Для каждой технологии существует свой драйвер, и приложению нужен тот самый, предназначенный для конкретного типа сервера, к которому подключается приложение.

JDBC-драйвер обеспечивает соединение с СУБД. Первый вариант — использовать его непосредственно, так что приложение будет запрашивать соединение с базой данных всякий раз, когда потребуется выполнить новую операцию с хранившимися там сведениями. Подобный метод часто встречается в учебниках по основам Java. Если вам доведется изучать JDBC по такому учебнику, вы, скорее всего, встретите в примерах класс под названием `DriverManager`, используемый для установки соединения, как показано в следующем фрагменте кода:

```
Connection con = DriverManager.getConnection(url, username, password);
```

В методе `getConnection()` используется URL, передаваемый в метод как значение первого параметра. Этот URL идентифицирует базу данных, нужную приложению, а также имя и пароль для аутентификации доступа к ней (рис. 12.4). Но постоянно запрашивать новое соединение и выполнять проверку для каждой операции — бессмысленная трата ресурсов и времени клиента и сервера баз данных. Представьте себе, что вы зашли в бар и заказали пиво. Вы выглядите молодо, поэтому бармен попросил вас показать паспорт. Это нормально. Но если бармен станет спрашивать паспорт, когда вы закажете второй и третий бокал (разумеется, гипотетически), это начнет раздражать.

```
Connection con = DriverManager.getConnection(url, username, password);
```

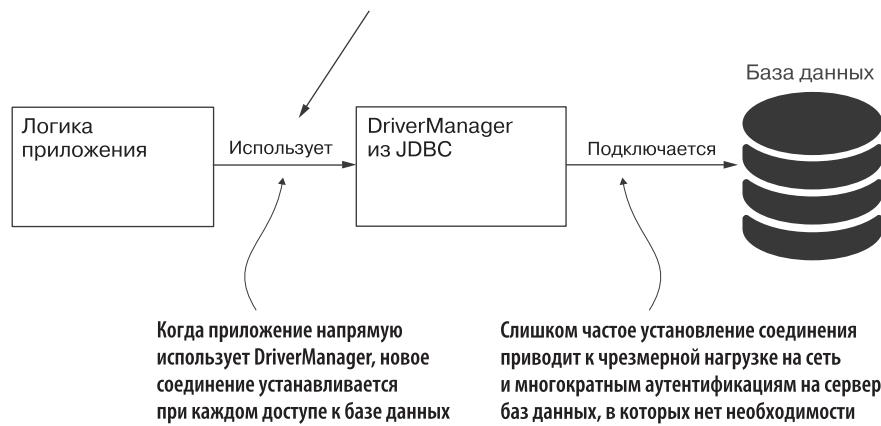


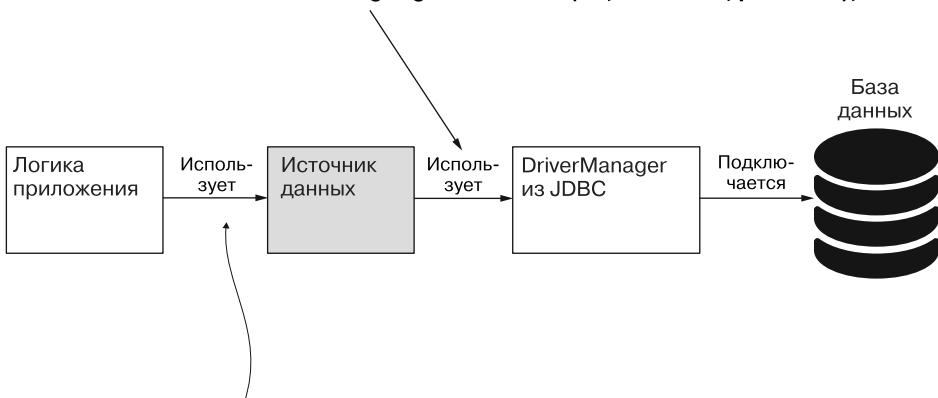
Рис. 12.4. Приложение может многократно использовать одно и то же соединение с сервером баз данных. Если устанавливать новые подключения без необходимости, приложение станет менее производительным, так как будет выполнять лишние операции. Для экономии соединений приложению нужен объект, отвечающий за управление ими, — источник данных

Объект источника данных эффективно управляет соединениями и сводит к минимуму количество ненужных операций. Вместо того чтобы напрямую использовать менеджер JDBC-драйвера, теперь для установки соединений и управления ими будет использоваться источник данных (рис. 12.5).

ПРИМЕЧАНИЕ

Источник данных — это объект, обязанность которого состоит в управлении соединениями приложения с сервером баз данных. Источник данных гарантирует успешность подключения и повышает производительность операций на уровне хранения данных.

```
Connection con = DriverManager.getConnection(url, username, password);
```



Источник данных управляет соединениями. При необходимости он подключает приложение к серверу баз данных и гарантирует, что новые соединения будут создаваться только тогда, когда это действительно нужно

Рис. 12.5. Добавляя в структуру классов источник данных, мы экономим время за счет того, что не выполняем лишние операции. Источник данных управляет соединениями, устанавливая подключения приложения к базе данных по мере необходимости, и создает новые соединения только тогда, когда это нужно

Для Java-приложений существует множество вариантов реализации источников данных, но в настоящее время чаще всего используется HikariCP (Hikari connection pool — пул соединений Hikari). Он является программным обеспечением с открытым кодом, так что вы тоже можете принять участие в его разработке.

12.2. ВЗАИМОДЕЙСТВИЕ С СОХРАНЕННЫМИ ДАННЫМИ С ПОМОЩЬЮ JDBCTEMPLATE

Далее мы создадим наше первое Spring-приложение, которое будет использовать базу данных, и рассмотрим преимущества, предоставляемые Spring для создания уровня хранения данных. Благодаря источнику данных наше приложение сможет более эффективно устанавливать соединения с базой. Но как с минимальными усилиями написать код для работы с данными? Мы убедились, что использование классов JDBC из JDK не особенно удобный способ взаимодействия с сохраненной информацией — даже для выполнения простейших операций приходится писать многословные блоки кода.

В примерах из учебников по основам Java вам мог встретиться код наподобие этого:

```
String sql = "INSERT INTO purchase VALUES (?,?)";
try (PreparedStatement stmt = con.prepareStatement(sql)) {
    stmt.setString(1, name);
    stmt.setDouble(2, price);
    stmt.executeUpdate();
} catch (SQLException e) {
    // сделать что-то в случае исключения
}
```

Так много кода, чтобы всего лишь добавить запись в таблицу! А ведь я еще убрал логику из блока `catch`. Но Spring поможет нам максимально сократить код, который нужно написать для выполнения подобных операций. В Spring-приложениях есть множество альтернативных вариантов для реализации уровня хранения данных; самые важные из них мы рассмотрим в главах 13 и 14. В этом разделе мы будем использовать инструмент, который называется `JdbcTemplate` и который позволяет упростить взаимодействие с базой данных посредством JDBC.

`JdbcTemplate` — самый простой из инструментов Spring, предназначенных для работы с реляционными базами данных. Но он отлично подходит для небольших приложений, так как не требует применения специализированных фреймворков для хранения информации. `JdbcTemplate` — лучшее из средств, предлагаемых Spring для реализации уровня хранения данных, если вы не хотите вводить в приложение дополнительные зависимости. Я также считаю, что это отличная отправная точка, чтобы научиться использовать уровень хранения данных в Spring-приложениях.

Рассмотрим, как работает `JdbcTemplate`, на примере. Для этого выполним следующие операции.

1. Установим соединение с СУБД.
2. Напишем логику для хранилища данных.
3. Вызовем методы хранилища из методов, реализующих действия для конечных точек REST.

Данный пример находится в проекте `sq-ch12-ex1`.

Для этого приложения мы создадим в базе данных таблицу `purchase`. В ней хранятся сведения о товарах, приобретенных в онлайн-магазине, а также о стоимости покупок. Таблица состоит из следующих столбцов (рис. 12.6):

- `id` — автоматически увеличивающееся уникальное значение, играющее роль первичного ключа таблицы;
- `product` — наименование купленного товара;
- `price` — стоимость покупки.

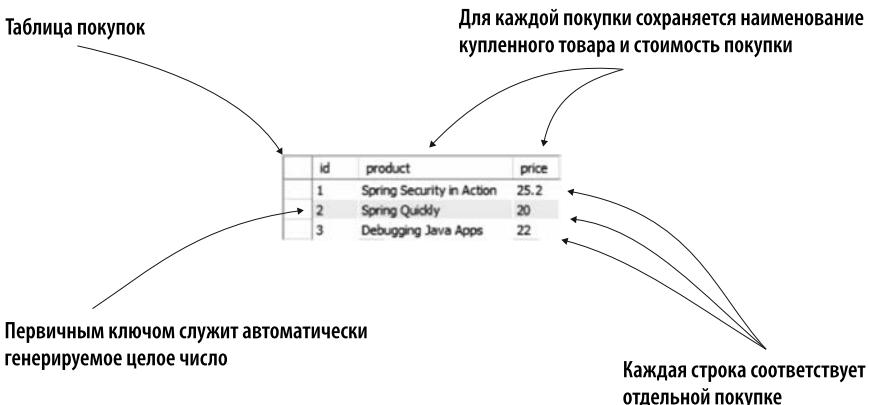


Рис. 12.6. Таблица покупок. Каждая покупка оформляется в виде строки таблицы. Для каждой сохраняются следующие атрибуты: наименование купленного товара и его стоимость. Первичным ключом таблицы (ID) является автоматически генерируемое число

Примеры, рассматриваемые в этой книге, не зависят от конкретной технологии реляционных баз данных — вы можете использовать данный код для любой из них. Но для выполнения примеров я должен был выбрать какую-то определенную. Нам понадобится H2 (база данных, хранящаяся в оперативной памяти: отлично подходит для примеров и, как станет ясно в главе 15, для интеграционных тестов) и MySQL (бесплатная и нетребовательная к ресурсам СУБД, легко устанавливается на локальных компьютерах и позволяет убедиться, что примеры работают не только для баз данных в оперативной памяти). Вы можете использовать любую другую технологию по вашему выбору: Postgres, Oracle или MS SQL. В таком случае при выполнении приложения необходимо будет использовать соответствующий JDBC-драйвер (как уже говорилось ранее и как вы знаете из основ Java). Кроме того, синтаксис SQL-запросов может различаться для разных технологий реляционных баз данных. Если вы выберете другую технологию, вам понадобится адаптировать SQL-запросы к ней.

ПРИМЕЧАНИЕ

Для базы данных H2 тоже нужен JDBC-драйвер. Но для нее этот драйвер не приходится добавлять специально, он поставляется в комплекте с зависимостью, которую мы добавим в файл pom.xml.

Составляя примеры для этого издания, я исходил из предположения, что вы уже знакомы с основами SQL и понимаете синтаксис простых SQL-запросов. Я также рассчитываю на то, что вам уже приходилось работать с JDBC как минимум при выполнении теоретических примеров, которые рассматриваются при изучении основ Java — обязательного условия для освоения Spring. Но если вы захотите

освежить свои знания в этой области, прежде чем двигаться дальше, советую прочесть главу 21 книги, посвященной JDBC, — Jeanne Boyarsky, Scott Selikoff, *OCP Oracle Certified Professional Java SE 11 Developer Complete Study Guide* (Sybex, 2020). Чтобы обновить представление о SQL, рекомендую 3-е издание книги Алана Болье «Изучаем SQL».

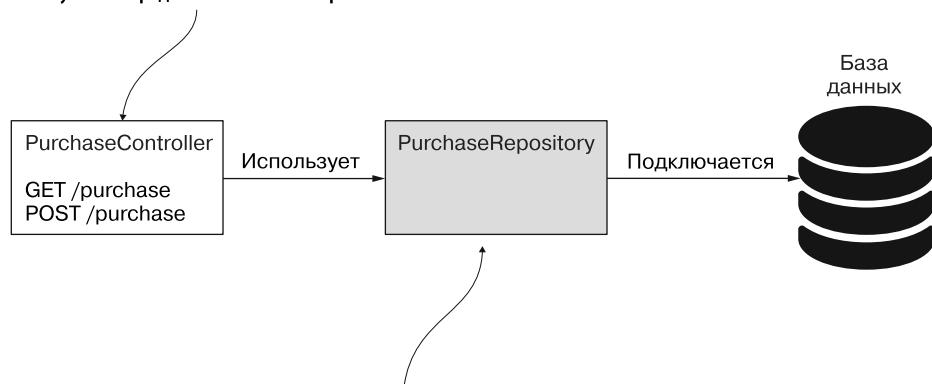
Требования к создаваемому нами приложению очень просты. Мы разработаем сервис бэкенда, у которого есть две конечные точки. Клиенты будут вызывать одну из них, чтобы добавить в таблицу покупок новую запись, и вторую, чтобы получить все записи из таблицы.

Работая с базой данных, мы представим все возможности уровня хранения данных в виде классов, которые (по соглашению) называются *репозиториями*. Структура классов создаваемого нами приложения показана на рис. 12.7.

ПРИМЕЧАНИЕ

Репозиторий — это класс, отвечающий за взаимодействие с базой данных.

PurchaseController — это REST-контроллер. Он предоставляет доступ к двум конечным точкам. Клиенты добавляют новые записи о покупках посредством вызова POST /purchase и получают все существующие в базе данных записи о покупках посредством вызова GET /purchase



PurchaseRepository использует предоставляемый Spring инструмент **JdbcTemplate**. С помощью источника данных **JdbcTemplate** подключается к серверу баз данных посредством JDBC

Рис. 12.7. В REST-контроллере реализованы две конечные точки. Когда клиент вызывает одну из них, контроллер делегирует управление объекту репозитория, который обращается к базе данных

Как обычно, начнем с того, что добавим все необходимые зависимости. В следующем фрагменте кода показано, какие зависимости нужно внести в файл `rom.xml`:

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
</dependency>
<scope>runtime</scope> ←
    </dependency>

```

Используем ту же веб-зависимость, что и в предыдущих главах, для создания конечных точек REST

Добавляем JDBC-диспетчер для получения доступа ко всем функциям, необходимым для взаимодействия с базами данных посредством JDBC

Добавляем зависимость H2, чтобы установить базу данных в оперативной памяти, используемую в данном примере, и JDBC-драйвер для доступа к ней

База данных и JDBC-драйвер понадобятся только во время работы приложения, но не на этапе компиляции. Чтобы сообщить Maven, что эти зависимости нужны только при выполнении приложения, добавляем тег `<scope>` со значением `runtime`

Зависимость H2 эмулирует базу данных на тот случай, если у вас нет сервера баз данных для выполнения примера. H2 – отличный инструмент, который можно использовать как в примерах, так и при тестировании приложений, когда необходимо проверить функционал продукта, исключив при этом его зависимость от базы данных (мы рассмотрим тестирование приложений в главе 15).

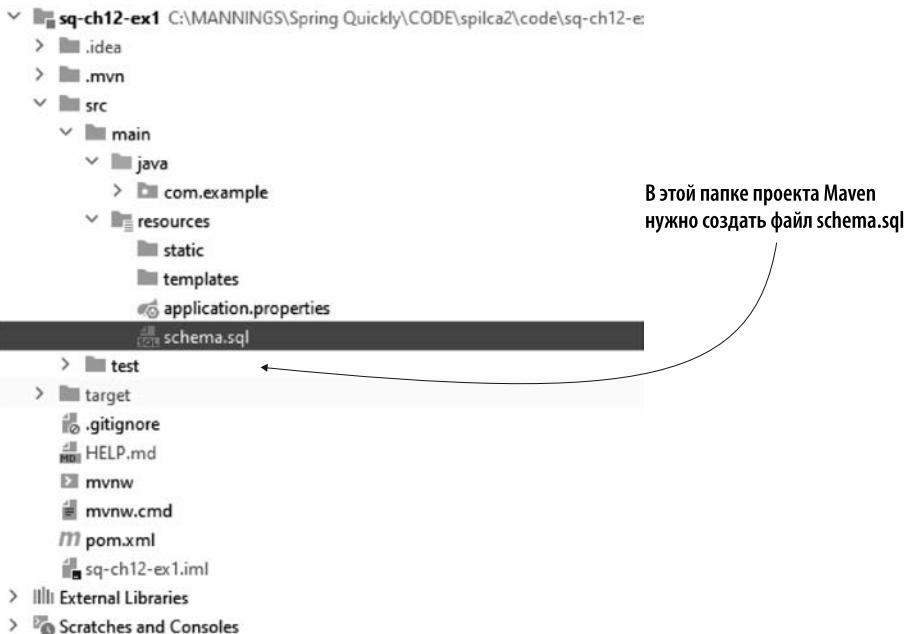


Рис. 12.8. В папке ресурсов проекта Maven нужно создать файл `schema.sql`, в который будут записаны запросы, определяющие структуру базы данных. Spring выполнит эти запросы при запуске приложения

Нам нужно сформировать таблицу, в которой будут храниться записи о покупках. В теоретических примерах структура базы данных создается легко — достаточно добавить файл `schema.sql` в папку ресурсов проекта Maven (рис. 12.8).

В этот файл мы запишем все структурообразующие SQL-запросы, то есть запросы, определяющие структуру базы данных. Некоторые разработчики называют их языком описания данных (data description language, DDL). Такой файл будет и у нас в проекте. Добавим туда следующий запрос для создания таблицы покупок:

```
CREATE TABLE IF NOT EXISTS purchase (
    id INT AUTO_INCREMENT PRIMARY KEY,
    product varchar(50) NOT NULL,
    price double NOT NULL
);
```

ПРИМЕЧАНИЕ

Описание структуры базы данных в файле `schema.sql` подходит только для теоретических примеров. Этот простой и быстрый способ помогает сосредоточиться на предмете, которому посвящен учебник, и не отвлекаться на определение структуры базы данных. Но на практике вам понадобится использовать зависимость, позволяющую управлять версиями скриптов для взаимодействия с базой данных. Советую обратить внимание на две такие очень популярные зависимости — Flyway (<https://flywaydb.org/>) и Liquibase (<https://www.liquibase.org/>). Их изучение выходит за рамки основ Spring, поэтому я не буду использовать их в примерах своей книги. Но советую ознакомиться с ними после того, как вы освоите базу по Spring.

Чтобы описать данные о покупке в приложении, нужно создать класс модели. Экземпляры этого класса соответствуют строкам таблицы покупок в базе данных. Поэтому у каждого из них должны быть атрибуты, где записан ID, наименование товара и цена. В следующем фрагменте кода представлен класс модели `Purchase`:

```
public class Purchase {

    private int id;
    private String product;
    private BigDecimal price;
    // Геттеры и сеттеры
}
```

Возможно, вас заинтересовал тот факт, что атрибут `price` класса `Purchase` имеет тип `BigDecimal`. Почему мы не объявили его как `double`? Здесь есть один важный момент, на который я хотел бы обратить ваше внимание: в теоретических примерах для представления дробных значений часто используют тип `double`, однако на практике применение `double` или `float` для дробных чисел не лучшая идея. Работая со значениями типа `double` и `float`, можно потерять точность даже при выполнении простых арифметических операций, таких как сложение

и вычитание. Все дело в том, в какой форме Java хранит эти значения в памяти. При работе с такой важной информацией, как цены, требуется `BigDecimal`. В Spring есть все функции, необходимые для его использования, поэтому не беспокойтесь о преобразовании типов.

ПРИМЕЧАНИЕ

Чтобы не потерять десятичную точность при выполнении различных операций, для хранения значений с плавающей точкой применяйте `BigDecimal` вместо `double` или `float`.

Чтобы легко получить в контроллере экземпляр `PurchaseRepository`, мы сделаем данный объект бином в контексте Spring. Для этого, как вы узнали из главы 3, проще всего использовать стереотипную аннотацию (такую как `@Component` или `@Service`).

Но в нашем случае вместо `@Component` мы воспользуемся специальной стереотипной аннотацией Spring для репозиториев — `@Repository`. В главе 3 вы познакомились с аннотацией `@Service`, которую мы использовали для классов сервисов; теперь мы добавим бин в контекст Spring с помощью `@Repository`. Определение класса репозитория представлено в листинге 12.1.

Листинг 12.1. Определение бина PurchaseRepository

```
@Repository ←
public class PurchaseRepository {
```

Добавляем бин этого типа в контекст
Spring с помощью стереотипной
аннотации `@Repository`

}

После того как бин `PurchaseRepository` добавлен в контекст приложения, можно внедрить экземпляр `JdbcTemplate`, который мы будем использовать для взаимодействия с базой данных. Я знаю, о чём вы сейчас подумали! «Откуда мы возьмем `JdbcTemplate`? Кто создал этот экземпляр, чтобы мы сейчас могли внедрить его в репозиторий?» В данном примере, как и во многих реальных приложениях, мы снова прибегнем к магии Spring Boot. Когда мы добавили в файл `root.xml` зависимость H2, Spring Boot автоматически настроил источник данных и создал экземпляр `JdbcTemplate`. В данном примере мы будем использовать непосредственно и то и другое.

При применении Spring без Spring Boot необходимы бины типа `DataSource` и `JdbcTemplate` (их можно добавить в контекст Spring с помощью аннотации `@Bean` в классе конфигурации, как было показано в главе 2). В разделе 12.3 я расскажу, как изменить эти бины, в каких случаях придется прописать собственные экземпляры источника данных и `JdbcTemplate`. В листинге 12.2 показано, как внедрить экземпляра `JdbcTemplate`, созданный Spring Boot для нашего приложения.

Листинг 12.2. Внедрение бина, созданного Spring Boot для работы с сохраненными данными

```
@Repository
public class PurchaseRepository {

    private final JdbcTemplate jdbc;

    public PurchaseRepository( ←
        JdbcTemplate jdbc) {
        this.jdbc = jdbc;
    }
}
```

Используем внедрение в конструкторе,
чтобы получить экземпляр JdbcTemplate
из контекста приложения

Итак, у нас есть экземпляр `JdbcTemplate`; теперь можно создать приложение в соответствии с заданными требованиями. У `JdbcTemplate` есть метод `update()`, позволяющий выполнять запросы по изменению данных — `INSERT`, `UPDATE` и `DELETE`. Чтобы выполнить такой запрос, достаточно передать в метод код и параметры SQL-запроса, остальное `JdbcTemplate` сделает сам (установит соединение, создаст SQL-оператор, обработает `SQLException` и т. д.). В листинге 12.3 показано, как добавить метод `storePurchase()` в класс `PurchaseRepository`. В `storePurchase()` мы с помощью `JdbcTemplate` создаем новую запись в таблице покупок.

Листинг 12.3. Создание записи в таблице с помощью `JdbcTemplate`

```
@Repository
public class PurchaseRepository {

    private final JdbcTemplate jdbc;

    public PurchaseRepository(JdbcTemplate jdbc) {
        this.jdbc = jdbc;
    }

    public void storePurchase(Purchase purchase) { ←
        String sql = ←
            "INSERT INTO purchase VALUES (NULL, ?, ?);";
        jdbc.update(sql, ←
            purchase.getProduct(),
            purchase.getPrice());
    }
}
```

В качестве параметра
метод принимает данные,
которые нужно сохранить

Запрос представляет собой
строку, в которой вместо
значений параметров стоят
вопросительные знаки (?).
Вместо ID ставим NULL, так как
СУБД сама генерирует значение
для этого столбца

Метод `update()` экземпляра `JdbcTemplate` посылает запрос
на сервер баз данных. Первый параметр метода — сам
запрос, а остальные — значения параметров запроса. Эти
значения в указанной последовательности подставляются
в запрос вместо вопросительных знаков

Написав всего пару строк кода, мы теперь можем вставлять, изменять и удалять записи в таблицах. Но с получением данных ситуация сложнее. Как и при создании записи, для этого нужно написать и отправить на сервер запрос — в нашем случае это соответственно `SELECT`. Но потом, чтобы сообщить `JdbcTemplate`,

как преобразовать данные в объекты `Purchase` (наш класс модели), необходим `RowMapper` — объект, выполняющий преобразование строки из `ResultSet` в заданный объект. Например, чтобы получить данные из базы и представить их в виде модели `Purchase`, необходимо создать `RowMapper` и описать в нем способ преобразования строки таблицы в экземпляр `Purchase` (рис. 12.9).

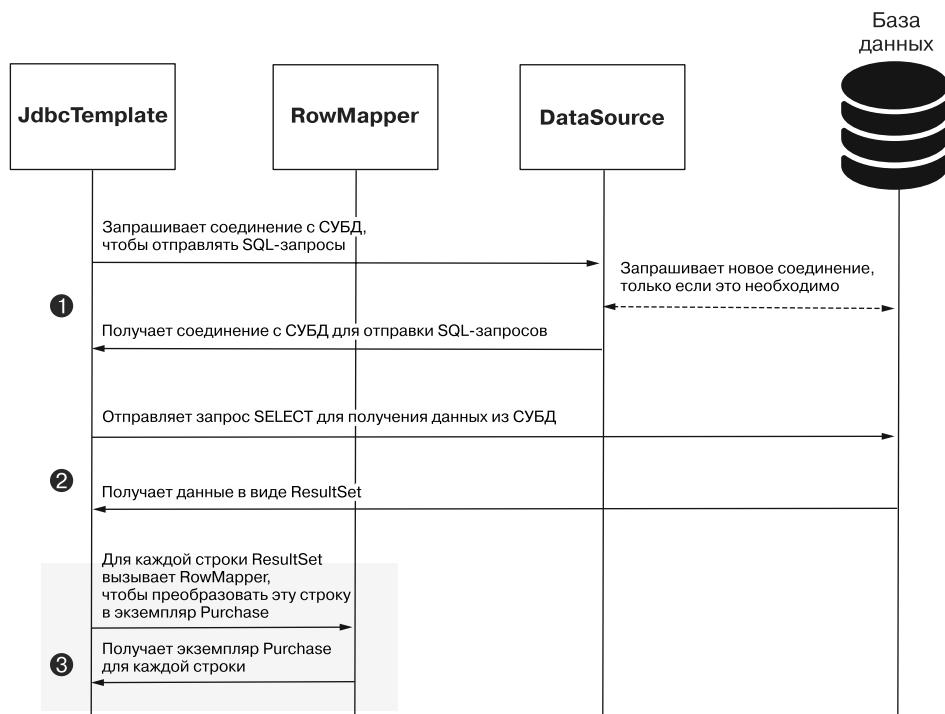


Рис. 12.9. С помощью `RowMapper` `JdbcTemplate` преобразует `ResultSet` в список экземпляров `Purchase`. Для каждой строки `ResultSet` `JdbcTemplate` вызывает `RowMapper`, преобразующий строку в экземпляр `Purchase`. На диаграмме показаны все три операции, которые выполняет `JdbcTemplate` при передаче запроса `SELECT`: 1 — установка соединения с СУБД, 2 — отправка запроса и получение результата и 3 — преобразование результата в объекты `Purchase`

В листинге 12.4 показано, как реализовать модель репозитория, чтобы извлечь все записи из таблицы покупок.

Теперь, когда мы написали методы репозитория и знаем, как сохранять записи в базе данных и получать их оттуда, можно открыть доступ к этим методам через конечные точки. Реализация контроллера представлена в листинге 12.5.

Листинг 12.4. Выборка записей из базы данных с помощью JdbcTemplate

```

Метод возвращает записи, полученные из базы
данных, в виде списка объектов Purchase

@Repository
public class PurchaseRepository {
    // Остальной код
    public List<Purchase> findAllPurchases() {
        String sql = "SELECT * FROM purchase";
        RowMapper<Purchase> purchaseRowMapper = (r, i) -> {
            Purchase rowObject = new Purchase();
            rowObject.setId(r.getInt("id"));
            rowObject.setProduct(r.getString("product"));
            rowObject.setPrice(r.getBigDecimal("price"));
            return rowObject;
        };
        return jdbc.query(sql, purchaseRowMapper);
    }
}

```

Определяем запрос SELECT для получения всех записей из таблицы покупок

Создаем объект RowMapper, который сообщает JdbcTemplate, как преобразовывать строку, полученную из базы данных, в объект Purchase. Параметр г лямбда-выражения соответствует ResultSet (данным, полученным из базы), а параметр i — целое число, показывающее номер строки

Заносим данные в экземпляр Purchase. JdbcTemplate будет выполнять эту логику для каждой строки из набора результатов

Отправляем запрос SELECT, используя метод query(), и передаем объект преобразователя строк, чтобы JdbcTemplate знал, как преобразовать полученные данные в объекты Purchase

Листинг 12.5. Использование объекта репозитория в классе контроллера

```

@RestController
@RequestMapping("/purchase")
public class PurchaseController {
    private final PurchaseRepository purchaseRepository;
    public PurchaseController(
        PurchaseRepository purchaseRepository) {
        this.purchaseRepository = purchaseRepository;
    }
    @PostMapping
    public void storePurchase(@RequestBody Purchase purchase) {
        purchaseRepository.storePurchase(purchase);
    }
    @GetMapping
    public List<Purchase> findPurchases() {
        return purchaseRepository.findAllPurchases();
    }
}

```

Используя внедрение зависимости в конструктор, извлекаем объект репозитория из контекста Spring

Создаем конечную точку, которую клиент может вызвать, чтобы сохранить в базе данных запись о покупке. Для сохранения данных, извлеченных действием контроллера из HTTP-запроса, используем метод storePurchase() репозитория

Создаем конечную точку, которую вызывает клиент, чтобы получить все записи из таблицы покупок. Действие контроллера извлекает данные из базы с помощью метода репозитория и затем возвращает их клиенту в теле HTTP-ответа

Запустив приложение, можно протестировать эти две конечные точки через Postman или cURL.

Чтобы добавить запись в таблицу покупок, используйте путь `/purchase` и HTTP-метод POST следующим образом:

```
curl -XPOST 'http://localhost:8080/purchase' \
-H 'Content-Type: application/json' \
-d '{
    "product" : "Spring Security in Action",
    "price" : 25.2
}'
```

Затем, вызвав конечную точку `/purchase` с HTTP-методом GET, можно убедиться, что приложение правильно сохранило запись о покупке. Вот команда cURL для этого запроса:

```
curl 'http://localhost:8080/purchase'
```

Тело HTTP-ответа на этот запрос представляет собой список всех записей о покупках, хранящихся в базе данных:

```
[
  {
    "id": 1,
    "product": "Spring Security in Action",
    "price": 25.2
  }
]
```

12.3. ИЗМЕНЕНИЕ КОНФИГУРАЦИИ ИСТОЧНИКА ДАННЫХ

Теперь давайте научимся изменять источник данных, используемый `JdbcTemplate` для взаимодействия с базой данных. База данных H2, которую мы применяли в разделе 12.2, отлично подходит для примеров и учебных пособий, а также на начальных стадиях создания уровня хранения данных в приложении. Но в реальных продуктах вам понадобится нечто большее, чем база в оперативной памяти. Кроме того, вам часто придется вносить изменения в источник данных.

Чтобы показать, как используются СУБД в реальных условиях, мы модифицируем пример, созданный в разделе 12.2, таким образом, чтобы задействовать MySQL-сервер. Как вы заметите, логику это не затронет — потребуется изменить лишь источник данных, который будет ссылаться на другую базу данных, и это нетрудно сделать. Нужно выполнить следующие операции.

1. В подразделе 12.3.1 мы добавим JDBC-драйвер MySQL и изменим конфигурацию источника данных с помощью файла `application.properties` таким образом, чтобы источник ссылался на базу данных MySQL. Мы по-прежнему

будем полагаться на Spring Boot, который автоматически поместит бин `DataSource` в контекст Spring в соответствии с заданными нами свойствами.

2. В подразделе 12.3.2 мы внесем изменения в проект, определив собственный бин `DataSource`, и подумаем, в каких случаях это может пригодиться на практике.

12.3.1. Определение источника данных в файле свойств приложения

Для начала подключим приложение к СУБД MySQL. В реальных приложениях используются внешние серверы баз данных, так что этот навык вам еще пригодится.

Проект, используемый здесь для демонстрации, называется `sq-ch12-ex2`. Если вы захотите выполнить пример самостоятельно (и я рекомендую это сделать), вам понадобится установить сервер MySQL и создать базу данных, к которой вы будете подключаться. Вы также можете при желании адаптировать этот пример для другой технологии (такой как Postgres или Oracle).

Для внесения изменений нам нужно выполнить следующее.

1. Изменить зависимости проекта, убрав оттуда H2 и добавив соответствующий JDBC-драйвер.
2. Добавить в файл `application.properties` свойства для соединения с новой базой данных.

Чтобы выполнить пункт 1, откройте файл `pom.xml` и удалите оттуда зависимость H2. Если вы намерены использовать MySQL, нужно добавить JDBC-драйвер MySQL. Сейчас в проекте понадобятся следующие зависимости:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

Добавляем JDBC-драйвер MySQL
 как зависимость, которая
 подключается при запуске
 приложения

Для выполнения пункта 2 нужно привести файл `application.properties` в соответствие со следующим фрагментом кода. Добавим в этот файл свойство `spring.datasource.url`, чтобы указать местонахождение базы данных,

а также свойства `spring.datasource.username` и `spring.datasource.password`, соответствующие параметрам доступа приложения для аутентификации и установки соединения с СУБД. Нам понадобится свойство `spring.datasource.initialization-mode` со значением `always`, чтобы Spring Boot использовал файл `schema.sql` для создания таблицы покупок. Для H2 нам это свойство не было нужно, так как в случае H2 Spring Boot по умолчанию выполняет запросы из `schema.sql`, если этот файл существует:

```
Указываем URL, соответствующий местоположению базы данных
spring.datasource.url=jdbc:mysql://localhost/spring_quickly?
useLegacyDatetimeCode=false&serverTimezone=UTC ←

spring.datasource.username=<dbms username> | Настраиваем параметры доступа
spring.datasource.password=<dbms password> | для аутентификации и соединения с СУБД
spring.datasource.initialization-mode=always ←

Включаем режим инициализации always, чтобы
Spring Boot выполнил запросы из файла schema.sql |
```

ПРИМЕЧАНИЕ

Хранить в файле свойств секретные данные, например пароли, — не лучшая идея для реальных приложений. Подобная конфиденциальная информация должна содержаться в закрытых хранилищах. Мы не будем обсуждать их в нашей книге, поскольку эта тема выходит за рамки базовых сведений. Но мне бы хотелось, чтобы вы помнили, что определение паролей подобным способом подходит только для примеров и учебных пособий.

После этих небольших изменений приложение использует базу данных MySQL. На основании свойств `spring.datasource`, предоставленных в `application.properties`, Spring Boot сможет создавать бин `DataSource`. Можно запустить приложение и проверить работу конечных точек, как мы это сделали в разделе 12.2.

Чтобы добавить новую запись в таблицу покупок, вызовите путь `/purchase` с HTTP-методом POST:

```
curl -XPOST 'http://localhost:8080/purchase' \
-H 'Content-Type: application/json' \
-d '{
    "product" : "Spring Security in Action",
    "price" : 25.2
}'
```

Затем можно вызвать конечную точку с путем `/purchase` и HTTP-методом GET, дабы убедиться, что приложение правильно сохранило запись о покупке. Команда cURL для этого запроса выглядит так:

```
curl 'http://localhost:8080/purchase'
```

Тело HTTP-ответа представляет собой список всех записей о покупках, сохраненных в базе данных:

```
[  
  {  
    "id": 1,  
    "product": "Spring Security in Action",  
    "price": 25.2  
  }  
]
```

12.3.2. Использование нестандартного бина DataSource

Чтобы Spring Boot смог использовать бин `DataSource`, нужно предоставить информацию о соединении в файле `application.properties`. Иногда этого достаточно, и я, как обычно, советую выбирать простейшее решение задачи. Но вы не всегда сможете рассчитывать на то, что Spring Boot создаст `DataSource` автоматически. Следовательно, придется сделать это самостоятельно. Вот ситуации, в которых может потребоваться создать бин `DataSource` вручную:

- необходима специфическая реализация `DataSource`, в зависимости от условий, возникающих в процессе выполнения приложения;
- приложение подключается к нескольким базам данных, так что нужно создать несколько источников данных и различать их посредством префиксов;
- необходимо определить специфические параметры объекта `DataSource` в некоторых случаях в процессе выполнения приложения — например, в зависимости от среды, из которой запускается приложение, пул соединений для оптимизации производительности может содержать больше или меньше подключений к базе данных;
- приложение использует фреймворк Spring без Spring Boot.

Ничего страшного здесь нет! `DataSource` — обычный бин, который добавляется в контекст Spring так же, как и любой другой. Вместо того чтобы предоставить Spring Boot автоматически создать и настроить конфигурацию объекта `DataSource`, вы можете определить в классе конфигурации метод с аннотацией `@Bean` (вы научились это делать в главе 3) и самостоятельно добавить объект в контекст. Тогда вы сможете полностью контролировать процесс его создания.

Модифицируем проект `sq-ch12-ex2` таким образом, чтобы бин источника данных не создавался автоматически посредством Spring Boot, а был определен вами. Эти изменения вы найдете в проекте `sq-ch12-ex3`. Мы создадим файл конфигурации и определим в нем метод с аннотацией `@Bean`. Этот метод будет возвращать экземпляр `DataSource`, который мы добавим в контекст Spring.

Класс конфигурации и определение метода с аннотацией `@Bean` показаны в следующем листинге.

Листинг 12.6. Определение бина DataSource для нашего проекта

```
@Configuration
public class ProjectConfig {

    @Value("${custom.datasource.url}")
    private String datasourceUrl;

    @Value("${custom.datasource.username}")
    private String datasourceUsername;

    @Value("${custom.datasource.password}")
    private String datasourcePassword;

    @Bean // Ставим перед методом аннотацию @Bean, чтобы
          // Spring добавил возвращаемое значение в контекст
    public DataSource dataSource() {
        HikariDataSource dataSource = // В качестве источника данных в этом примере
                                      // мы будем использовать HikariCP. Но если проект
                                      // требует чего-то другого, самостоятельно создавая бин,
                                      // вы можете выбрать любой другой источник данных
                                      new HikariDataSource();

        dataSource.setJdbcUrl(datasourceUrl);
        dataSource.setUsername(datasourceUsername);
        dataSource.setPassword(datasourcePassword); // Устанавливаем параметры
        dataSource.setConnectionTimeout(1000); // соединения для источника данных

        return dataSource; // Вы можете определить и другие параметры
    } // (которые, возможно, понадобятся при определенных
      // условиях). В данном случае я в качестве примера
      // использовал время ожидания подключения (сколько
      // времени источник данных будет ждать установки
      // соединения, прежде чем решит, что оно не удалось)

    } // Возвращаем экземпляр DataSource,
      // который Spring внесет в контекст
}
```

Параметры соединения могут изменяться, поэтому имеет смысл и дальше указывать их отдельно от кода приложения. В данном примере они хранятся в файле application.properties

Метод возвращает объект DataSource. Если Spring Boot обнаруживает, что в контексте Spring уже есть DataSource, новый он не создает

Ставим перед методом аннотацию @Bean, чтобы Spring добавил возвращаемое значение в контекст

В качестве источника данных в этом примере мы будем использовать HikariCP. Но если проект требует чего-то другого, самостоятельно создавая бин, вы можете выбрать любой другой источник данных

Устанавливаем параметры соединения для источника данных

Вы можете определить и другие параметры (которые, возможно, понадобятся при определенных условиях). В данном случае я в качестве примера использовал время ожидания подключения (сколько времени источник данных будет ждать установки соединения, прежде чем решит, что оно не удалось)

Не забудьте определить значения свойств, которые вы внедряете, с помощью аннотации `@Value`. В файле `application.properties` эти свойства должны выглядеть так, как показано в следующем фрагменте. Я намеренно использовал слово `custom` в их именах, чтобы подчеркнуть, что свойства не определяет Spring Boot и что мы сами выбираем имена. Вы можете присвоить любые:

```
custom.datasource.url=jdbc:mysql://localhost/spring_quickly?
useLegacyDatetimeCode=false&serverTimezone=UTC
```

```
custom.datasource.username=root
custom.datasource.password=
```

Теперь можно запустить проект `sq-ch12-ex3` и проверить его работу. Результаты должны быть такими же, как и для двух предыдущих проектов в этой главе.

Чтобы добавить новую запись в таблицу покупок, вызываем путь `/purchase` с HTTP-методом POST:

```
curl -XPOST 'http://localhost:8080/purchase' \
-H 'Content-Type: application/json' \
-d '{
    "product" : "Spring Security in Action",
    "price" : 25.2
}'
```

Затем можно вызвать конечную точку `/purchase` с HTTP-методом GET, чтобы убедиться, что приложение правильно сохранило запись о покупке. Команда cURL для этого запроса выглядит так:

```
curl 'http://localhost:8080/purchase'
```

Тело HTTP-ответа на запрос представляет собой список всех записей о покупках, хранящихся в базе данных:

```
[  
  {  
    "id": 1,  
    "product": "Spring Security in Action",  
    "price": 25.2  
  }  
]
```

ПРИМЕЧАНИЕ

Если вы не очищали таблицу покупок и использовали ту же базу данных, что и для проекта sq-ch12-ex2, результат будет содержать и записи, внесенные в базу данных ранее.

РЕЗЮМЕ

- Абстракции объектов, необходимых для соединения Java-приложения с реляционной базой данных, содержатся в пакете Java Development Kit (JDK). Реализация этих абстракций находится в зависимости, которую всегда нужно подключать на этапе выполнения приложения. Эта зависимость называется JDBC-драйвером.
- Источник данных — это объект, управляющий соединениями с сервером баз данных. Без источника данных приложению требовалось бы устанавливать такие подключения слишком часто, что снижало бы производительность.
- По умолчанию Spring Boot выбирает источник данных под названием HikariCP. Используя пул соединений, этот источник данных оптимизирует применение приложением соединений с базой данных. Вы можете предпочтеть другие технологии, если они лучше подходят для вашего продукта.
- `JdbcTemplate` — это инструмент Spring, позволяющий упростить код, который необходимо написать для доступа к реляционной базе данных посредством

JDBC-драйвера. Для соединения с сервером баз данных `JdbcTemplate` использует источник данных.

- Для отправки запроса, изменяющего данные в таблице, применяется метод `update()` объекта `JdbcTemplate`. Чтобы получить данные посредством отправки запросов `SELECT`, используется один из методов `query()` объекта `JdbcTemplate`. Как правило, такие операции нужны для изменения или получения сохраненных данных.
- Чтобы создать свой источник данных вместо того, который автоматически предлагает Spring Boot, нужен бин типа `java.sql.DataSource`. Если объявить бин этого типа в контексте Spring, то Spring Boot будет использовать его, а не опцию по умолчанию. То же самое происходит, если необходимо создать нестандартный объект `JdbcTemplate`. Как правило, мы применяем те объекты, которые Spring Boot определяет по умолчанию, но иногда встречаются ситуации, когда для различных оптимизаций необходимы специальные конфигурации или реализации объектов.
- Если приложение должно соединяться с несколькими базами данных, можно создать несколько объектов — источников данных, по одному для каждого `JdbcTemplate`. В этом случае, чтобы различать объекты одного типа в контексте приложения, необходимо использовать аннотацию `@Qualifier` (вы уже научились это делать в главах 4 и 5).

13

Транзакции в Spring-приложениях

В этой главе

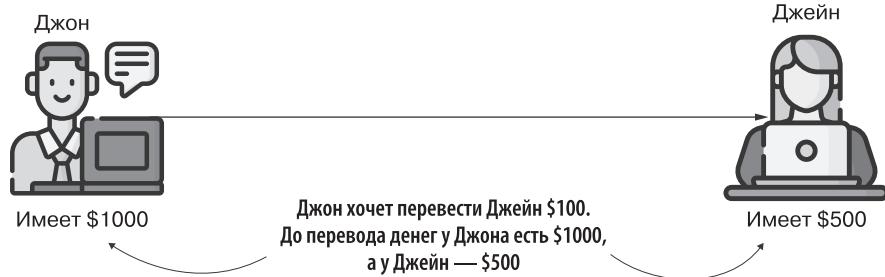
- ✓ Что такое транзакция.
- ✓ Как Spring управляет транзакциями.
- ✓ Использование транзакций в Spring-приложении.

Один из самых важных моментов, которые следует учитывать при управлении данными, — это сохранение их точности. Нам бы не хотелось, чтобы выполнение какого-либо сценария привело к появлению неверных или несогласованных данных. Покажу на примере. Предположим, вы разрабатываете приложение для перевода денег — электронный кошелек. В этом приложении у пользователей есть счета, где они хранят деньги. Вы создаете функционал, позволяющий переводить средства с одного счета на другой. В качестве примера рассмотрим простейшую реализацию этой опции, которая состоит из двух операций (рис. 13.1).

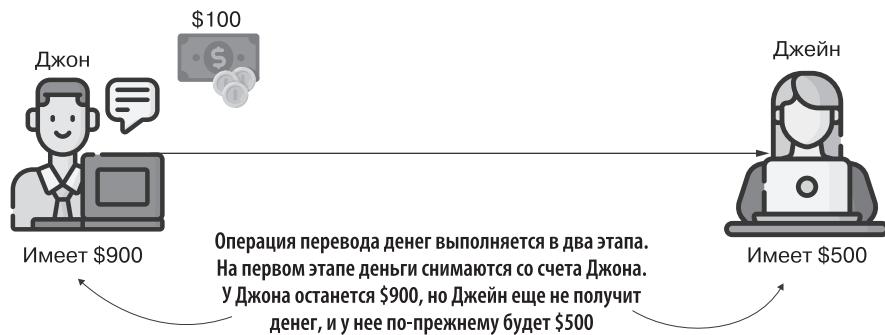
1. Снять деньги со счета отправителя.
2. Положить деньги на счет получателя.

Обе эти операции изменяют данные (изменяющие операции), и для правильно-го перевода денег обе должны завершиться успешно. Но что случится, если во время выполнения второй операции возникнет проблема и она не выполнится? Если первая операция будет проведена, а вторая — нет, то данные перестанут быть согласованными.

До операции перевода денег



Этап 1. Снять \$100 со счета Джона



Этап 2. Положить \$100 на счет Джейн

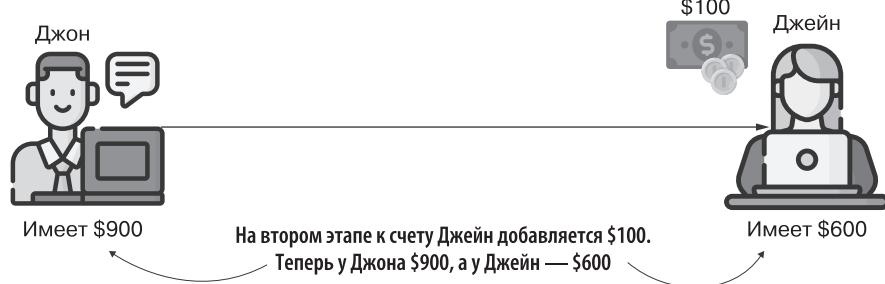
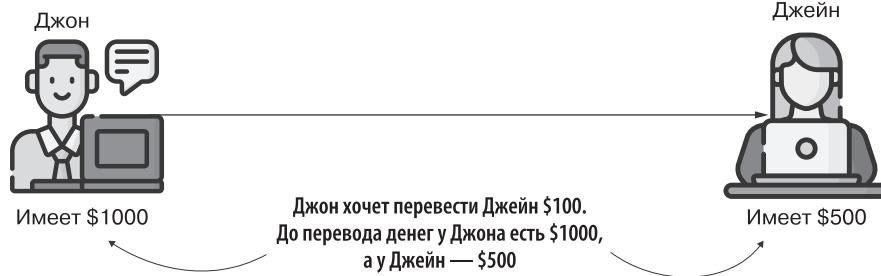


Рис. 13.1. Пример сценария использования. При переводе денег с одного счета на другой приложение выполняет две операции: снимает переводимую сумму с первого счета и добавляет ее к средствам второго. Мы запрограммируем данный сценарий; при этом нужно гарантировать, что в процессе выполнения не нарушится согласованность данных

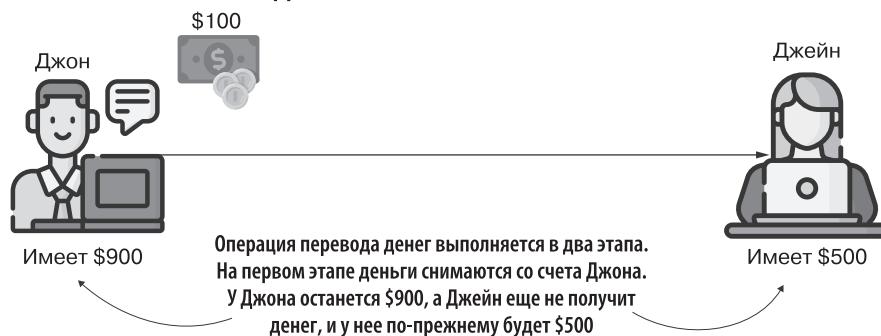
Предположим, Джон переводит Джейн \$100. До перевода на счету Джона есть \$1000, а у Джейн — \$500. После того как перевод завершится, мы ожидаем, что на счету Джона станет на \$100 меньше (то есть $\$1000 - \$100 = \$900$), а у Джейн — на \$100 больше: $\$500 + \$100 = \$600$.

Если вторая операция завершится неудачно, возникнет ситуация, при которой деньги были сняты со счета Джона, но так и не дошли до Джейн. У Джона останется \$900, а у Джейн по-прежнему будет \$500. Куда девались \$100? Это поведение программы показано на рис. 13.2.

До операции перевода денег



Этап 1. Снять \$100 со счета Джона



Этап 2. Перевод \$100 на счет Джейн не удался



Рис. 13.2. Если одна из операций сценария использования завершится неудачно, то данные перестанут быть согласованными. В случае перевода денег, если снятие со счета отправителя выполнится успешно, а зачисление на счет получателя окажется неудачным, деньги будут потеряны

Во избежание подобных ситуаций, когда нарушается согласованность данных, необходимо гарантировать, что будут проведены либо обе операции, либо ни одна из них. Транзакции предоставляют возможность сделать так, чтобы несколько операций либо выполнялись правильно, либо не выполнялись вовсе.

13.1. ТРАНЗАКЦИИ

Далее мы рассмотрим *транзакции*. Транзакция — это определенный набор изменяющих операций, которые либо выполняются корректно все, либо не выполняются вообще. Подобное явление называется *атомарностью*. Транзакции — необходимая часть приложений, они гарантируют согласованность данных в случае, если один из этапов сценария использования, выполняемый после изменения данных, завершится неудачно. Снова рассмотрим (упрощенную) процедуру перевода денег, которая состоит из двух операций.

1. Снять деньги со счета отправителя.
2. Положить деньги на счет получателя.

Мы начнем транзакцию перед п. 1 и завершим ее после п. 2 (рис. 13.3). Тогда, если оба этапа будут выполнены успешно, при завершении транзакции (после п. 2) приложение сохранит изменения, внесенные на обоих этапах. В этом случае также принято говорить, что транзакция будет «зафиксирована». «Фиксация» имеет место в конце транзакции в том случае, если все операции были удачными и приложение может сохранить измененные данные.

ФИКСАЦИЯ

Успешное завершение транзакции, при котором приложение сохраняет все изменения, сделанные изменяющими операциями в процессе транзакции.

Если п. 1 выполнится без проблем, но п. 2 по какой-то причине завершится неудачно, приложение вернется к состоянию, в котором оно находилось до работы с п. 1. Эта операция называется *откатом*.

ОТКАТ

Если приложение возвращает данные к тому состоянию, в котором они находились в начале транзакции во избежание нарушения согласованности данных, то говорят, что транзакция завершается откатом.

13.2. ТРАНЗАКЦИИ В SPRING

Прежде чем я покажу вам, как использовать транзакции в Spring-приложении, рассмотрим, как транзакции работают в Spring и какие возможности предлагает фреймворк для построения их кода. Собственно говоря, транзакции в Spring выполняются на основе аспекта Spring AOP. (Мы изучали работу аспектов в главе 6.)

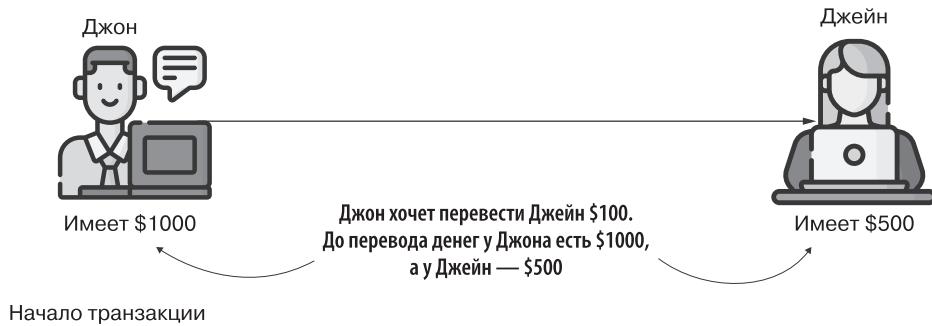
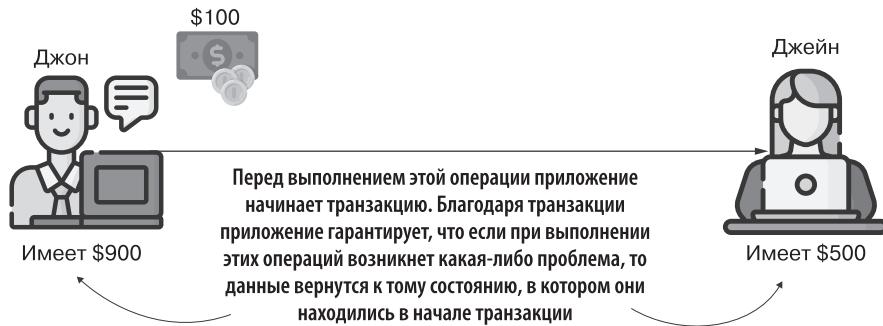
До операции перевода денег**Этап 1. Снять \$100 со счета Джона****Этап 2. Перевод \$100 на счет Джейн не удался**

Рис. 13.3. Транзакция позволяет избежать несогласованности данных, которая может возникнуть, если одна из операций сценария использования не будет выполнена. Благодаря этому, если один из этапов завершится неудачно, данные вернутся в то состояние, в котором они находились перед началом транзакции

Аспект — это код, который вмешивается в работу конкретных методов способом, определяемым программистом. Сегодня методы, выполнение которых должно перехватываться и изменяться посредством аспектов, обычно отмечаются аннотациями. В случае транзакций в Spring происходит то же самое. Чтобы отметить метод, который должен быть задействован в рамках транзакции, используется аннотация `@Transactional`. При этом Spring невидимо для нас настраивает конфигурацию аспекта (вы не должны создавать его; за вас это сделает Spring) и применяет логику транзакции для тех операций, которые выполняются в данном методе (рис. 13.4).

Spring знает, что если при выполнении метод выбросит исключение, то транзакцию нужно откатить. Хочу подчеркнуть: именно метод должен выбросить исключение. Когда я читаю лекции по Spring, студенты часто считают достаточными исключения, возникающие при проведении какой-либо операции внутри метода `transferMoney()`. Но это не так! Чтобы аспект понял, что нужно откатить изменения, метод транзакции должен передать исключение дальше. Если метод этого не сделает, а лишь сам обработает исключение своей логикой, аспект так и не узнает, что исключение возникло (рис. 13.5).

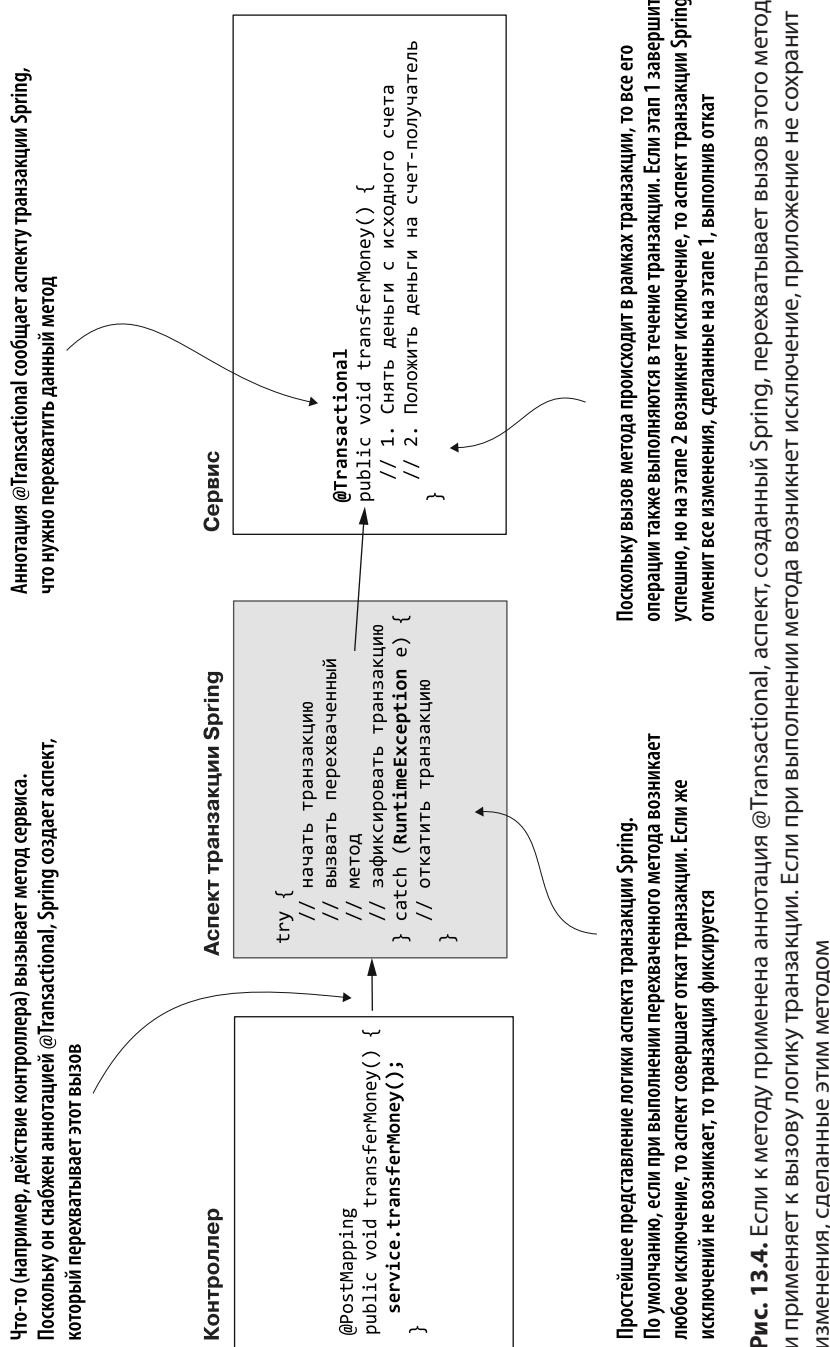
А КАК БЫТЬ С ПРОВЕРЯЕМЫМИ ИСКЛЮЧЕНИЯМИ В ТРАНЗАКЦИЯХ?

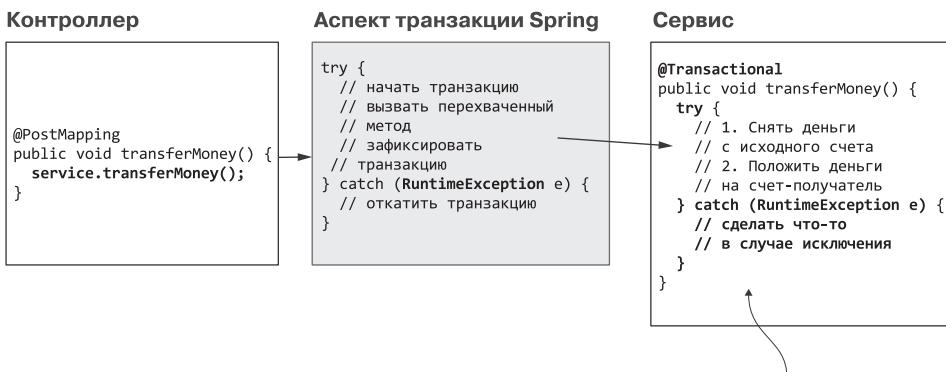
До сих пор мы говорили только об исключениях, возникающих при выполнении приложения. А как быть с проверяемыми исключениями? Проверяемыми исключениями в Java называют такие исключения, которые нужно обработать или выбросить, иначе приложение не будет скомпилировано. Если метод выдаст подобное исключение, приведет ли это тоже к откату транзакции? По умолчанию — нет! Spring откатывает транзакцию автоматически только в случае исключения при выполнении. Это поведение большинства реальных приложений.

В ситуациях с проверяемым исключением используется условие `throws` в сигнатуре метода, поэтому если они возникают, то не являются причиной, которая может привести к несогласованности данных. Наоборот, это контролируемый процесс, который должен управляться логикой, написанной разработчиком.

Но если вы хотите, чтобы Spring в случае проверяемых исключений тоже откатывал транзакции, вы можете изменить поведение фреймворка, предусмотренное по умолчанию. У аннотации `@Transactional`, которую вы научитесь использовать в разделе 13.3, есть атрибуты, позволяющие определить, какие исключения должны приводить к откату транзакций в Spring.

Однако я советую вам всегда по возможности делать приложение как можно более простым и полагаться на стандартное поведение фреймворка, если только не будет особых причин поступить иначе.





Если при выполнении одной из операций внутри метода возникнет исключение, но метод сам обработает это исключение и не передаст его дальше вызывающему методу, аспект не получит исключение и зафиксирует транзакцию. При работе с исключениями в транзакционных методах, таких как этот, необходимо помнить, что аспект, управляющий транзакцией, не выполнит откат, если не увидит исключение

Рис. 13.5. Если внутри метода возникнет исключение выполнения, но метод сам обработает это исключение и не передаст его дальше вызывающему методу, аспект не получит исключение и зафиксирует транзакцию. При работе с исключениями в транзакционных методах, таких как этот, необходимо помнить, что аспект, управляющий транзакцией, не выполнит откат, если не увидит исключение

13.3. ИСПОЛЬЗОВАНИЕ ТРАНЗАКЦИЙ В SPRING-ПРИЛОЖЕНИЯХ

Начнем с примера, который научит вас использовать транзакции в Spring-приложении. Чтобы объявить транзакцию в приложении Spring, достаточно воспользоваться аннотацией `@Transactional` — ею отмечается метод, который Spring должен заключить в транзакцию. Больше ничего делать не нужно. Фреймворк сам создает аспект, который будет перехватывать методы с `@Transactional`. Этот аспект запускает транзакцию и либо фиксирует изменения, вносимые методом, если все заканчивается успешно, либо откатывает их, если при выполнении возникает исключение.

Мы напишем приложение, которое сохраняет информацию о состоянии счета в таблице базы данных. Предположим, что это бэкенд разрабатываемого нами электронного кошелька. Мы создадим функционал для перевода денег с одного счета на другой. Для реализации этого сценария использования нам понадобится создать транзакцию, чтобы гарантировать согласованность данных в случае, если возникнет исключение.

Структура классов нашего приложения очень проста. Мы будем хранить данные о счетах (включая количество денег) в таблице базы данных. Для работы с этими данными мы создадим репозиторий, а бизнес-логику (реализующую перевод денег) разместим в классе сервиса. Именно для метода сервиса, реализующего бизнес-логику, нам и понадобится транзакция. Чтобы открыть доступ к этому сценарию использования, мы создадим конечную точку в классе контроллера. Для перевода денег с одного счета на другой нужно будет ее вызывать. Структура классов нашего приложения показана на рис. 13.6.

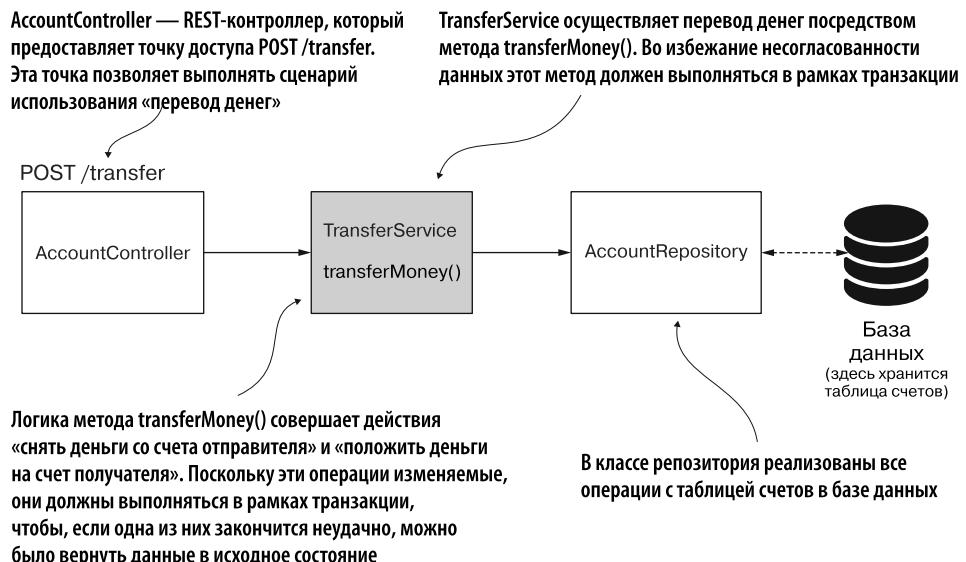


Рис. 13.6. Мы реализуем сценарий использования «перевод денег» в виде класса сервиса и сделаем этот сервис доступным через конечную точку REST. Для доступа к данным, хранящимся в базе данных, и для их изменения метод сервиса будет использовать класс репозитория. Во избежание несогласованности данных, которая может возникнуть в случае проблем при работе метода, метод сервиса (в котором реализована бизнес-логика) должен выполняться в рамках транзакции

Этот пример находится в проекте `sq-ch13-ex1`. Мы создадим проект Spring Boot и добавим в его файл `pom.xml` зависимости, как показано ниже. Мы продолжим использовать Spring JDBC (как делали это в главе 12) и базу данных H2, хранящуюся в оперативной памяти:

```

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

```

346 Часть II. Реализация

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Приложение взаимодействует всего с одной таблицей базы данных. Мы назовем эту таблицу `account`, и она будет содержать следующие поля:

- `id` — первичный ключ. Мы определим это поле как автоматически увеличивающееся значение типа INT;
- `name` — имя владельца счета;
- `amount` — сумма денег, которую владелец держит на этом счету.

Для создания таблицы воспользуемся файлом `schema.sql` в папке ресурсов проекта. Запишем туда следующий SQL-запрос для создания таблицы:

```
create table account (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    amount DOUBLE NOT NULL
);
```

Кроме `schema.sql`, мы разместим в папке ресурсов файл `data.sql`, чтобы сделать две записи, которые впоследствии понадобятся для тестирования приложения. В `data.sql` содержатся два SQL-запроса, создающие в базе данных две записи счетов. Эти запросы выглядят так:

```
INSERT INTO account VALUES (NULL, 'Helen Down', 1000);
INSERT INTO account VALUES (NULL, 'Peter Read', 1000);
```

Чтобы получить доступ к данным из приложения, нам нужен класс, моделирующий таблицу счетов. Поэтому мы создадим класс `Account`, который будет моделировать записи счетов в базе данных. Этот класс показан в листинге 13.1.

Листинг 13.1. Класс Account, моделирующий таблицу счетов

```
public class Account {

    private long id;
    private String name;
    private BigDecimal amount;

    // Геттеры и сеттеры
}
```

Чтобы реализовать сценарий использования «перевод денег», нам понадобятся следующие функции на уровне репозитория.

1. Поиск информации о счете по ID этого счета.
2. Изменение баланса для заданного счета.

Мы реализуем эти функции с помощью `JdbcTemplate`, как было показано в главе 10. Для выполнения пункта 1 создадим метод `findAccountById(long id)`, который получает ID счета в виде параметра и благодаря `JdbcTemplate` берет из базы данных информацию о счете с заданным ID. Для выполнения пункта 2 нам понадобится метод `changeAmount(long id, BigDecimal amount)`. Он присваивает сумму, которую получает в виде второго параметра, счету, ID которого является первым параметром метода. Реализация этих двух методов показана в листинге 13.2.

Листинг 13.2. Реализация функций хранения данных в репозитории

```
@Repository
public class AccountRepository {
    private final JdbcTemplate jdbctemplate;

    public AccountRepository(JdbcTemplate jdbctemplate) {
        this.jdbctemplate = jdbctemplate;
    }

    public Account findAccountById(long id) {
        String sql = "SELECT * FROM account WHERE id = ?";
        return jdbctemplate.queryForObject(sql, new AccountRowMapper(), id);
    }

    public void changeAmount(long id, BigDecimal amount) {
        String sql = "UPDATE account SET amount = ? WHERE id = ?";
        jdbctemplate.update(sql, amount, id);
    }
}
```

С помощью аннотации `@Repository` добавляем бин этого класса в контекст Spring, чтобы затем внедрить в класс сервиса, где мы будем его использовать

С помощью метода `queryForObject()` объекта `JdbcTemplate` получаем информацию о счете, отправляя в СУБД запрос `SELECT`. Нам также понадобится `RowMapper`, чтобы сообщить `JdbcTemplate`, как преобразовать строку — результат запроса в объект модели

С помощью метода `update()` объекта `JdbcTemplate` изменяем баланс счета, отправляя в СУБД запрос `UPDATE`

Как вы уже знаете из главы 12, при использовании `JdbcTemplate` для получения информации из базы данных посредством запроса `SELECT` необходимо создать объект `RowMapper`, который покажет `JdbcTemplate`, как преобразовать каждую строку результата из базы данных в конкретную модель объекта. В данном случае нам нужно показать `JdbcTemplate`, как модифицировать строку в объект `Account`. Реализация `RowMapper` показана в листинге 13.3.

Листинг 13.3. Преобразование строки в экземпляр объекта модели посредством RowMapper

```

    Реализуем контракт RowMapper и передаем ему
    в качестве параметризованного типа класс модели,
    в которую будет преобразована строка результата

public class AccountRowMapper
    implements RowMapper<Account> { ←

    @Override
    public Account mapRow(ResultSet resultSet, int i) ←
        throws SQLException {
        Account a = new Account();
        a.setId(resultSet.getInt("id"));
        a.setName(resultSet.getString("name"));
        a.setAmount(resultSet.getBigDecimal("amount"));
        return a; ←
    } ←
} ←

```

Реализуем метод mapRow(), получающий в качестве параметра строку результата (в виде объекта ResultSet) и возвращающий экземпляр Account, в который преобразуется текущая строка

Преобразуем значения текущей строки результата в атрибуты Account

Возвращаем экземпляр Account, в который были преобразованы результаты выполнения запроса

Чтобы нам было проще проверить работу приложения, добавим еще одну функцию, позволяющую получить из базы данных информацию о счете, как показано в листинге 13.4. Мы будем использовать эту функцию для уверенности, что приложение работает как положено.

Листинг 13.4. Получение всех записей счетов из базы данных

```

@Repository
public class AccountRepository {

    // Остальной код

    public List<Account> findAllAccounts() {
        String sql = "SELECT * FROM account";
        return jdbc.query(sql, new AccountRowMapper());
    }
}

```

В классе сервиса мы реализовали логику сценария использования «перевод денег». Для управления данными в таблице счетов в классе TransferService использован класс AccountRepository. В методе реализована следующая логика.

- Получить информацию о счетах отправителя и получателя и извлечь оттуда баланс для каждого счета.
- Снять сумму перевода со счета отправителя. Для этого счету присваивается новое значение, представляющее собой старый баланс минус сумма перевода.

3. Положить сумму перевода на счет получателя. Для этого счету присваивается новое значение, представляющее собой старый баланс плюс сумма перевода.

В листинге 13.5 представлен метод `transferMoney()` класса сервиса, в котором реализована эта логика. Обратите внимание, что пункты 2 и 3 являются изменяющими операциями — обе изменяют сохраненные данные (баланс счета). Если выполнять их не в транзакции, то неудачное завершение одной из них может привести к несогласованности данных.

К счастью, чтобы отметить метод как транзакционный и сообщить Spring, что его выполнение необходимо перехватить и заключить в транзакцию, достаточно поставить перед методом аннотацию `@Transactional`. В листинге 13.5 показана реализация сценария использования «перевод денег» в классе сервиса.

Листинг 13.5. Реализация сценария использования «перевод денег» в классе сервиса

```

@Service
public class TransferService {

    private final AccountRepository accountRepository;

    public TransferService(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    @Transactional // С помощью аннотации @Transactional сообщаем Spring, что вызов
    // этого метода должен происходить в рамках транзакции
    public void transferMoney(long idSender,
        long idReceiver,
        BigDecimal amount) {
        Account sender =
            accountRepository.findAccountById(idSender);
        Account receiver =
            accountRepository.findAccountById(idReceiver); // Получаем информацию
                                                        // о счетах, чтобы извлечь
                                                        // оттуда текущий баланс
                                                        // для каждого счета

        BigDecimal senderNewAmount =
            sender.getAmount().subtract(amount); // Вычисляем новый баланс
                                                // для счета получателя
        BigDecimal receiverNewAmount =
            receiver.getAmount().add(amount); // Вычисляем новый баланс
                                                // для счета отправителя

        accountRepository
            .changeAmount(idSender, senderNewAmount); // Присваиваем новое значение баланса
                                                        // для счета отправителя

        accountRepository
            .changeAmount(idReceiver, receiverNewAmount); // Присваиваем новое значение
                                                        // баланса для счета получателя
    }
}

```

На рис. 13.7 визуально представлена область транзакции и операции, которые выполняет метод `transferMoney()`.

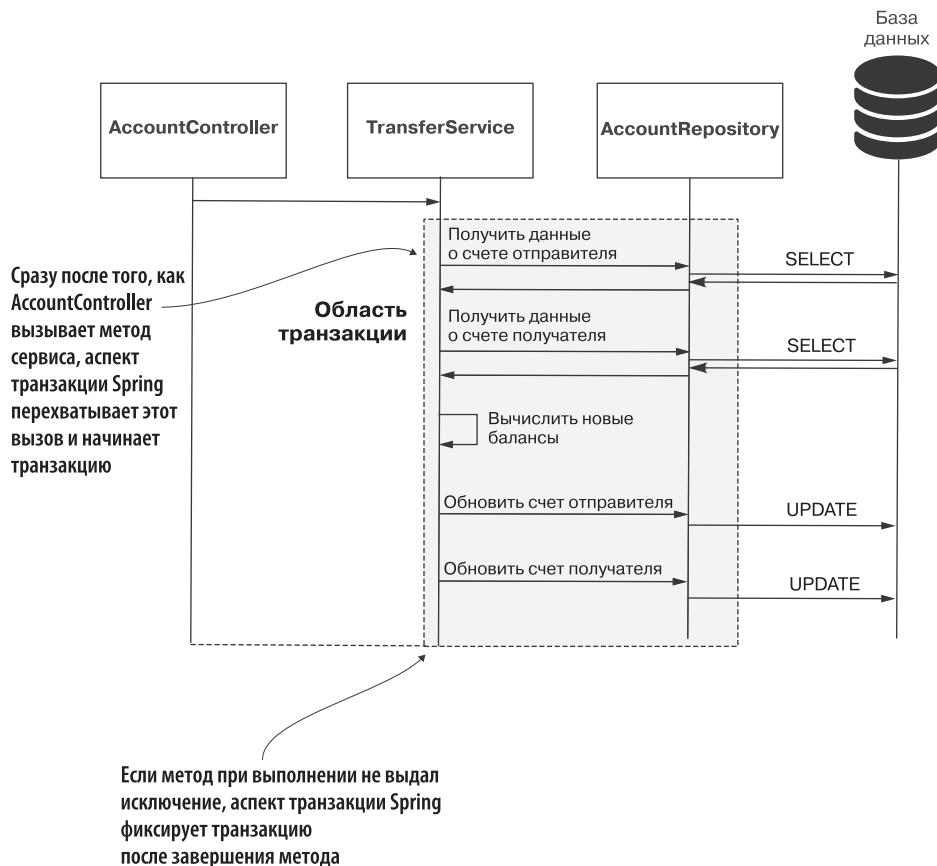


Рис. 13.7. Транзакция начинается непосредственно перед выполнением метода сервиса и заканчивается сразу после того, как этот метод успешно завершится. Если метод не выбрасывает исключение при выполнении, то приложение фиксирует транзакцию. Если на любом этапе возникает исключение при выполнении, приложение возвращает данные к тому состоянию, в котором они находились в начале транзакции

Напишем также метод, получающий данные для всех счетов. Мы предоставим доступ к нему посредством конечной точки в классе контроллера, который создадим позже. Мы будем применять этот метод при тестировании сценария использования «перевод денег», чтобы убедиться, что данные были изменены правильно.

ИСПОЛЬЗОВАНИЕ АННОТАЦИИ @TRANSACTIONAL

Аннотацию @Transactional можно использовать и для всего класса. Если аннотация относится к классу (как показано в следующем примере кода), то она применяется ко всем его методам. В реальных приложениях @Transactional часто применяют именно к классу, поскольку методы класса сервиса определяют сценарии использования и, как правило, все эти сценарии должны выполняться в рамках транзакций. Чтобы не ставить аннотацию перед каждым методом, проще сразу отметить ею весь класс. При использовании @Transactional и для класса, и для метода уровень метода переопределяет уровень класса:

```
@Service
@Transactional
public class TransferService {
    // Остальной код

    public void transferMoney(long idSender,
        long idReceiver,
        BigDecimal amount) {
        // Остальной код
    }
}
```

Аннотацию @Transactional часто применяют ко всему классу. Если в классе содержится несколько методов, то @Transactional используется для всех методов класса

В листинге 13.6 показана реализация метода getAllAccounts(), который возвращает список всех записей счетов, существующих в базе данных.

Листинг 13.6. Метод сервиса, возвращающий все существующие счета

```
@Service
public class TransferService {
    // Остальной код

    public List<Account> getAllAccounts() {
        return accountRepository.findAllAccounts();
    }
}
```

В листинге 13.7 представлена реализация класса AccountController, в котором определены конечные точки для доступа к методам сервиса.

Листинг 13.7. Доступ к сценариям использования через конечные точки REST в классе контроллера

```
@RestController
public class AccountController {
```

```

private final TransferService transferService;

public AccountController(TransferService transferService) {
    this.transferService = transferService;
}

@PostMapping("/transfer")
public void transferMoney(
    @RequestBody TransferRequest request
) {
    transferService.transferMoney(
        request.getSenderId(),
        request.getReceiverAccountId(),
        request.getAmount());
}

@GetMapping("/accounts")
public List<Account> getAllAccounts() {
    return transferService.getAllAccounts();
}
}

```

Используем для конечной точки /transfer HTTP-метод POST, так как вносим изменения в информацию из базы данных

Извлекаем из тела запроса необходимые значения (ID счета отправителя и счета получателя, а также сумму перевода)

Вызываем метод сервиса transferMoney() — транзакционный метод, в котором реализован сценарий перевода денег

В качестве параметра для действия контроллера `transferMoney()` мы будем использовать объект типа `TransferRequest`, моделирующий тело HTTP-запроса. Подобные объекты, обязанность которых состоит в моделировании данных, передаваемых между двумя приложениями, являются DTO. В листинге 13.8 показано определение DTO `TransferRequest`.

Листинг 13.8. Объект переноса данных `TransferRequest`, моделирующий тело HTTP-запроса

```

public class TransferRequest {

    private long senderAccountId;
    private long receiverAccountId;
    private BigDecimal amount;

    // Остальной код
}

```

Запустим приложение и проверим, как работает транзакция. Для вызова конечных точек, предоставляемых приложением, можно использовать cURL или Postman. Вначале вызовем конечную точку `/accounts`, чтобы посмотреть, как выглядят данные до перевода денег. Ниже представлена команда cURL для вызова конечной точки `/accounts`:

```
curl http://localhost:8080/accounts
```

После выполнения этой команды в консоли должен появиться следующий вывод:

```
[{"id":1,"name":"Helen Down","amount":1000.0}, {"id":2,"name":"Peter Read","amount":1000.0}]
```

В базе данных есть два счета (эти записи были внесены в базу ранее в этом разделе, когда мы создали файл `data.sql`). У Хелен и Питера есть по \$1000. Теперь выполним сценарий использования «перевод денег» и переведем Питеру от Хелен \$100. Для этого нужно вызвать конечную точку `/transfer` с помощью следующей команды cURL:

```
curl -XPOST -H "content-type:application/json" -d '{"senderAccountId":1, "receiverAccountId":2, "amount":100}' http://localhost:8080/transfer
```

Если теперь снова вызовем конечную точку `/accounts`, увидим, что суммы на счетах изменились. После перевода денег у Хелен осталось \$900, а у Питера теперь есть \$1100:

```
curl http://localhost:8080/accounts
```

Результат вызова конечной точки `/accounts` после перевода денег выглядит так:

```
[{"id":1,"name":"Helen Down","amount":900.0}, {"id":2,"name":"Peter Read","amount":1100.0}]
```

Приложение работает, и выполнение сценария использования приводит к ожидаемому результату. Но как убедиться, что транзакция действительно функционирует? Пока все идет хорошо, приложение правильно сохраняет данные, но как узнать, вернет ли оно данные к исходному состоянию, если что-то внутри метода приведет к исключению при выполнении? Следует ли просто принять это на веру? Конечно же, нет!

ПРИМЕЧАНИЕ

Одна из самых важных вещей, которые я узнал о приложениях, — никогда не верьте, что что-то работает, пока вы сами тщательно все не протестируете!

Мне нравится говорить, что, пока вы не протестируете некую функцию приложения, эта функция находится в состоянии Шредингера: прежде чем будет доказано ее точное состояние, она и работает, и не работает. Разумеется, это всего лишь моя личная аналогия с одной из ключевых концепций квантовой механики.

Проверим, работает ли откат транзакции так, как ожидается, в случае исключения при выполнении. Я создал копию проекта `sq-ch13-ex1` и назвал ее

sq-ch13-ex2. Я добавил туда всего одну строку кода в конце метода сервиса `transferMoney()` — эта строка выдает исключение при выполнении, как показано в листинге 13.9.

Листинг 13.9. Имитация проблемы, возникающей при выполнении сценария использования

```
@Service
public class TransferService {

    // Остальной код

    @Transactional
    public void transferMoney(
        long idSender,
        long idReceiver,
        BigDecimal amount) {

        Account sender = accountRepository.findAccountById(idSender);
        Account receiver = accountRepository.findAccountById(idReceiver);

        BigDecimal senderNewAmount = sender.getAmount().subtract(amount);
        BigDecimal receiverNewAmount = receiver.getAmount().add(amount);

        accountRepository.changeAmount(idSender, senderNewAmount);
        accountRepository.changeAmount(idReceiver, receiverNewAmount);

        throw new RuntimeException("Oh no! Something went wrong!");
    }
}                                В конце метода сервиса выбрасываем исключение при выполнении,
                                    чтобы имитировать проблему в процессе транзакции
```

На рис. 13.8 показано, какие изменения мы внесли в метод сервиса `transferMoney()`.

Запустим приложение и проверим данные о счетах, вызвав конечную точку `/accounts`, которая возвращает все записи о счетах, хранящиеся в базе данных:

```
curl http://localhost:8080/accounts
```

После выполнения этой команды в консоли должно появиться следующее:

```
[{"id":1,"name":"Helen Down","amount":1000.0},
 {"id":2,"name":"Peter Read","amount":1000.0}]
```

Как и в предыдущем teste, вызовем конечную точку `/transfer`, чтобы перевести Питеру от Хелен \$100. Для этого воспользуемся следующей командой cURL:

```
curl -XPOST -H "content-type:application/json" -d '{"senderAccountId":1,
  "receiverAccountId":2, "amount":100}' http://localhost:8080/transfer
```

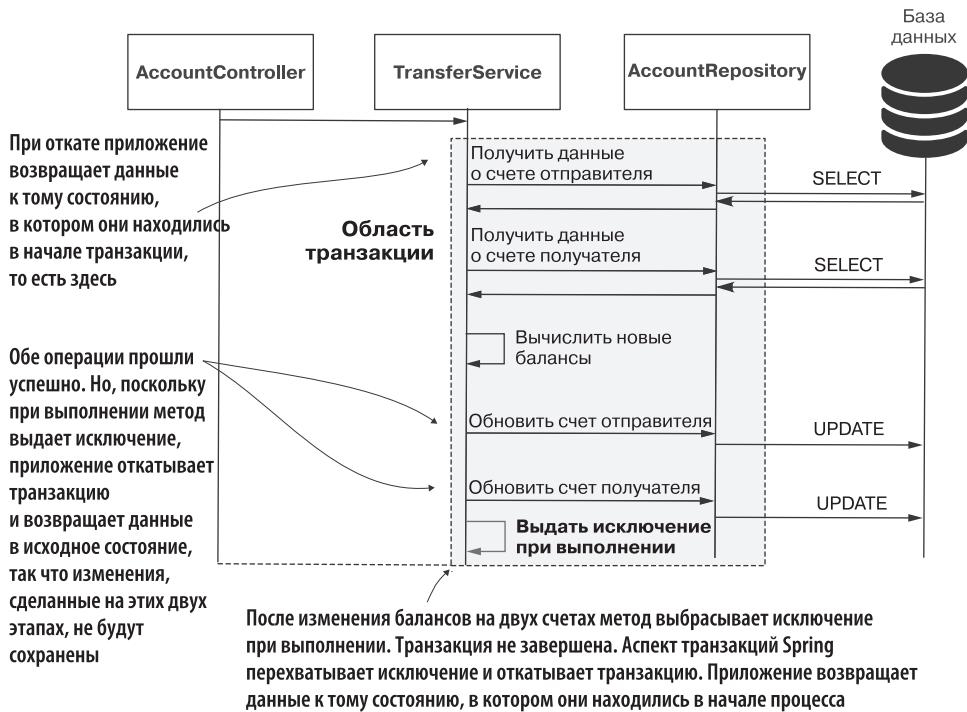


Рис. 13.8. Если метод выбрасывает исключение при выполнении, то Spring откатывает транзакцию. Все успешно выполненные изменения данных не сохраняются. Приложение возвращает данные к тому виду, в котором они находились в начале транзакции

Но теперь метод `transferMoney()` класса сервиса выбрасывает исключение, из-за чего в ответ на запрос клиент получает ошибку 500. Вы увидите это исключение в консоли приложения. Трассировка стека исключения выглядит приблизительно так:

```
java.lang.RuntimeException: Oh no! Something went wrong!
  at
com.example.services.TransferService.transferMoney(TransferService.java:30)
→ ~[classes/:na]
  at
com.example.services.TransferService$$FastClassBySpringCGLIB$$338bad6b.invoke
→ (<generated>) ~[classes/:na]
  at
org.springframework.cglib.proxy.MethodProxy.invoke(MethodProxy.java:218)
→ ~[spring-core-5.3.3.jar:5.3.3]
```

Снова вызовем конечную точку `/accounts` и посмотрим, изменились ли данные о счетах:

```
curl http://localhost:8080/accounts
```

После выполнения этой команды в консоли приложения должно появиться следующее:

```
[  
  {"id":1,"name":"Helen Down","amount":1000.0},  
  {"id":2,"name":"Peter Read","amount":1000.0}  
]
```

Как видим, данные остались теми же, несмотря на то что исключение возникло после выполнения двух операций, изменяющих балансы счетов. На счету Хелен должно было быть \$900, а на счету Питера — \$1100. Однако там остались те же суммы, что и раньше. Так произошло потому, что приложение выполнило откат транзакции, из-за чего данные вернулись к состоянию, в котором они находились в начале транзакции. Были выполнены обе операции, изменяющие данные, но потом аспект транзакций Spring получил исключение при выполнении — и все отменил.

РЕЗЮМЕ

- Транзакция — это набор операций, изменяющих данные. Эти операции либо выполняются все вместе, либо не выполняются вовсе. В реальных приложениях во избежание несогласованности данных практически любой сценарий использования должен быть представлен в виде транзакции.
- Если одна из операций завершается неудачно, приложение возвращает данные к тому виду, в котором они находились в начале транзакции. В таких случаях принято говорить, что происходит откат транзакции.
- Если все операции выполнены успешно, то говорят, что происходит фиксация транзакции. Это означает, что приложение сохраняет все изменения, совершенные в процессе выполнения сценария использования.
- Для реализации кода транзакции в Spring используется аннотация `@Transactional`. Она ставится перед методом, который должен выполняться в рамках транзакции. Аннотацию `@Transactional` также можно поставить перед классом, и тогда Spring будет считать транзакционными все его методы.
- При выполнении приложения методы с аннотацией `@Transactional` перехватываются специальным аспектом Spring. Этот аспект открывает транзакцию и, если в процессе проведения транзакции возникает исключение, откатывает ее. Если метод не выбрасывает исключение, транзакция фиксируется, и приложение сохраняет сделанные методом изменения.

11

Сохранение данных с помощью Spring Data

В этой главе

- ✓ Знакомство со Spring Data.
- ✓ Создание репозиториев Spring Data.
- ✓ Реализация уровня хранения данных в Spring с помощью Spring Data JDBC.

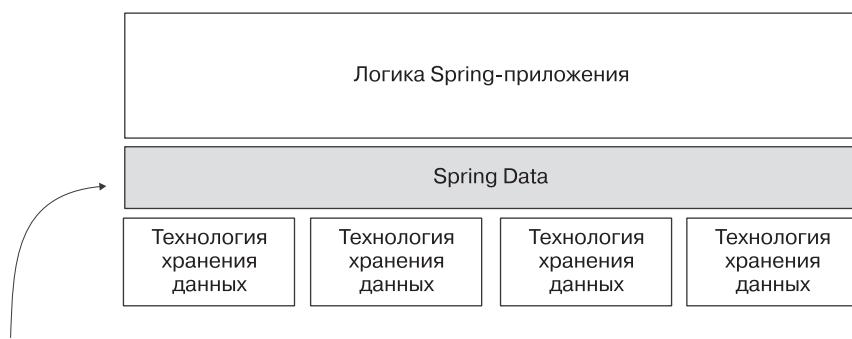
Пришла пора научиться использовать Spring Data — проект экосистемы Spring, который позволяет с минимальными усилиями реализовать уровень хранения данных в Spring-приложении. Как вы уже знаете, главное назначение фреймворка — предоставление готовых функций, которые можно сразу же использовать в приложении. Фреймворки экономят много времени и, кроме того, делают структуру приложений более понятной.

В этой главе вы будете создавать репозитории приложений путем объявления интерфейсов. Реализацию этих интерфейсов обеспечит фреймворк. Вы буквально сделаете так, что приложение сможет взаимодействовать с базой данных, но при этом вам не придется самим разрабатывать репозиторий, и вообще вы затратите минимум усилий.

Вначале мы рассмотрим, как работает Spring Data. В разделе 14.2 вы узнаете, как интегрировать Spring Data в приложения Spring. Затем, в разделе 14.3, мы перейдем к практическому примеру, на котором вы научитесь использовать Spring Data JDBC при настройке уровня хранения данных в приложении.

14.1. ЧТО ТАКОЕ SPRING DATA

Посмотрим, что такое Spring Data и зачем его нужно использовать в Spring-приложениях. Spring Data — это проект экосистемы Spring, который упрощает разработку уровня хранения данных, предоставляя реализацию необходимых объектов в соответствии с выбранной технологией хранения данных. Благодаря этому нам, чтобы определить репозитории Spring-приложения, остается написать лишь несколько строк кода. На рис. 14.1 предлагается визуальное представление места, которое занимает Spring Data в структуре приложения.



Spring Data — это надстройка, которая упрощает реализацию хранения данных путем унификации различных технологий хранения данных под общими абстракциями

Рис. 14.1. В экосистеме Java есть множество различных технологий хранения данных, каждая из которых применяется особым способом. У каждой из них — свои абстракции и структура классов. Spring Data обеспечивает общий уровень абстракций, расположенный над всеми этими технологиями и позволяющий упростить их использование

Посмотрим, какое место занимает Spring Data в Spring-приложении. Существуют различные технологии для взаимодействия приложения с сохраненными данными. В главах 12 и 13 мы использовали технологию JDBC, которая обеспечивает непосредственное соединение с реляционной СУБД через менеджер драйверов. Но JDBC не единственный способ подключиться к реляционной базе данных. Другой распространенный способ организации хранения информации — использование ORM-фреймворка, такого как Hibernate. Кроме того, реляционные базы данных не единственный тип технологии хранения сведений. Для реализации этого функционала в приложении может применяться также одна из технологий NoSQL.

На рис. 14.2 показаны некоторые возможности хранения данных в Spring. У каждой из них есть свой способ реализации репозиториев. У некоторых технологий (таких как JDBC) есть даже несколько вариантов работы с уровнем хранения данных в приложении. Например, как вы уже знаете из главы 12, в случае JDBC можно использовать `JdbcTemplate`, но можно и взаимодействовать непосредственно с интерфейсами JDK (`Statement`, `PreparedStatement`, `ResultSet` и др.). Такое разнообразие лишь добавляет сложностей.

Есть много способов внедрения уровня хранения данных. Приложение может соединяться с СУБД напрямую посредством JDBC либо использовать другие библиотеки для подключения к NoSQL-реализациям, таким как MongoDB, Neo4J и другие технологии хранения данных



Рис. 14.2. Соединение с реляционной СУБД посредством JDBC не единственный способ реализации уровня хранения данных в приложении. На практике вы будете использовать и другие варианты, причем для каждого из них есть своя библиотека и набор API, которые придется изучить. Такое многообразие сильно усложняет жизнь

Картина еще больше усложняется, если добавить в нее ORM-фреймворки, например Hibernate. На рис. 14.3 показано, какое место занимает Hibernate в этой схеме. Есть разные способы непосредственного использования JDBC в приложении, но вместо этого можно взять фреймворк, который является надстройкой над JDBC.

Но не стоит беспокоиться! Нет необходимости изучать все это сразу, и вы не должны все это знать, чтобы использовать Spring Data. Того, что мы обсудили в главах 12 и 13 о JDBC, достаточно, чтобы начать разбираться со Spring Data. Я сообщаю вам все это лишь затем, чтобы объяснить, почему Spring Data так важен. Возможно, вы уже задавались вопросом, существует ли единый способ реализации хранения данных для всех упомянутых выше технологий? Да, такой способ есть — и это как раз Spring Data.

JDBC может использоваться в Spring-приложении непосредственно или через ORM-фреймворк, например Hibernate

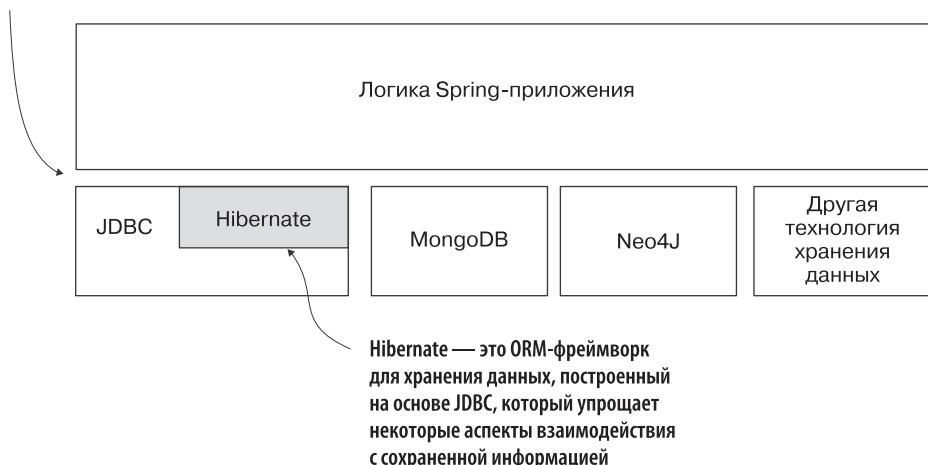


Рис. 14.3. В некоторых приложениях используются фреймворки, такие как Hibernate, которые являются надстройкой над JDBC. Их многочисленность делает реализацию уровня хранения данных еще более запутанной. Хотелось бы избежать этих сложностей в наших приложениях. Ниже вы узнаете, как Spring Data может помочь нам в этом

Spring Data упрощает реализацию уровня хранения данных следующими способами:

- предоставляет общий набор абстракций (интерфейсов) для различных технологий хранения данных. Это обеспечивает один и тот же подход при использовании этих технологий;
- позволяет пользователю создавать операции с сохраненными данными, используя только абстракции, реализации которых предоставляет Spring Data. Таким образом приходится писать меньше кода и можно быстрее настроить функции приложения. Кроме того, благодаря меньшему количеству строк приложение становится более понятным и его проще поддерживать.

На рис. 14.4 показано наглядно, какое место занимает Spring Data в Spring-приложении. Как видим, Spring Data — это высокуровневая надстройка над различными способами реализации хранения данных. Таким образом, какой бы вариант хранения данных вы ни выбрали, при использовании Spring Data операции с сохраненными данными будут выглядеть одинаково.

Spring Data — это высокоуровневая надстройка, которая упрощает реализацию хранения данных, так как объединяет различные технологии под одними и теми же абстракциями

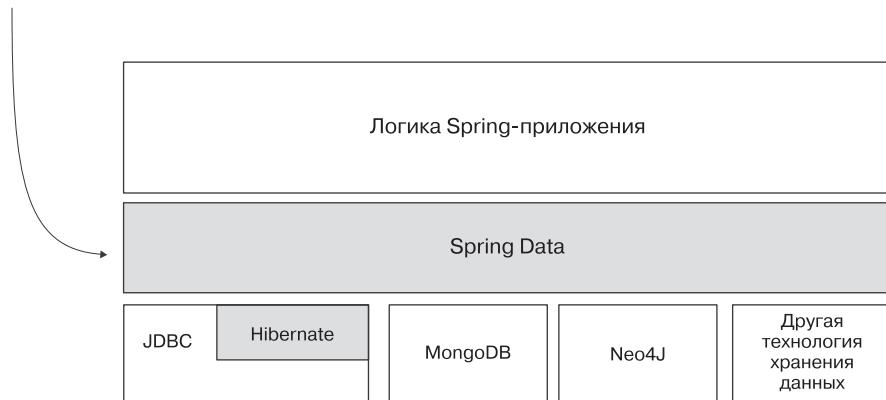


Рис. 14.4. Spring Data упрощает реализацию уровня хранения данных, предоставляя общий набор абстракций для разных технологий

14.2. КАК РАБОТАЕТ SPRING DATA

Теперь мы обсудим, как работает Spring Data и чем он может быть полезен при реализации уровня хранения данных в Spring. Говоря о Spring Data, разработчики обычно имеют в виду все возможности, предоставляемые этим проектом для работы Spring-приложения с той или иной технологией хранения данных. Но в конкретном приложении обычно используется только одна технология: JDBC, Hibernate, MongoDB или какая-то другая.

В Spring Data есть разные модули, предназначенные для соответствующих технологий. Эти модули не зависят друг от друга, и их можно подключать к проекту, используя различные зависимости Maven. Таким образом при разработке приложения вам *не нужна* зависимость Spring Data. Ее просто не существует. Spring Data предоставляет отдельные зависимости Maven для каждой поддерживаемой им технологии хранения данных. Например, для соединения с СУБД непосредственно через JDBC используется модуль Spring Data JDBC, а для подключения к базе данных MongoDB — Spring Data Mongo. На рис. 14.5 показано, как Spring Data взаимодействует с JDBC.

Полный список модулей Spring Data находится на официальной странице Spring Data — <https://spring.io/projects/spring-data>.

Приложение использует только одну из технологий хранения данных, поэтому необходимо подключить тот модуль Spring Data, который ей соответствует. Если в приложении используется JDBC, то нужна зависимость для модуля Spring Data JDBC

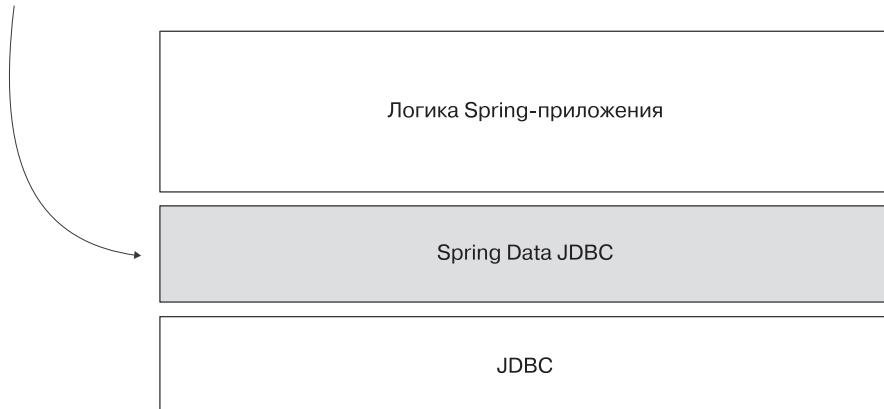


Рис. 14.5. Если в приложении используется JDBC, нужно подключить только ту часть проекта Spring Data, которая управляет хранением данных посредством JDBC. Модуль Spring Data, отвечающий за эту функцию, называется Spring Data JDBC. Чтобы добавить его в проект, нужно подключить зависимость этого модуля

Какая бы технология хранения данных ни использовалась в приложении, Spring Data предоставляет один и тот же набор интерфейсов (контрактов), которые нужно расширить, чтобы описать соответствующие функции. На рис. 14.6 представлены следующие интерфейсы:

- **Repository** — наиболее абстрактный контракт. Если его расширить, приложение распознает интерфейс, написанный как любой репозиторий Spring Data. Однако этот интерфейс не сможет унаследовать какие-либо предопределенные операции (такие как создание записи, чтение всех записей или чтение записи по первичному ключу). В интерфейсе **Repository** не объявлен ни один метод;
- **CrudRepository** — простейший контракт Spring Data, который, кроме всего прочего, предоставляет некоторые функции по взаимодействию с сохраненными данными. Если его расширить, чтобы эти функции определить, то станут доступны простейшие операции создания, чтения, изменения и удаления записей;
- **PagingAndSortingRepository** — расширяет **CrudRepository**, добавляя операции сортировки и чтения определенного количества записей (постранично).

Repository — это интерфейс-маркер.
Он не содержит методов, его назначение — лежать
в основе иерархии контрактов Spring Data. Скорее всего,
вы не будете расширять этот интерфейс непосредственно

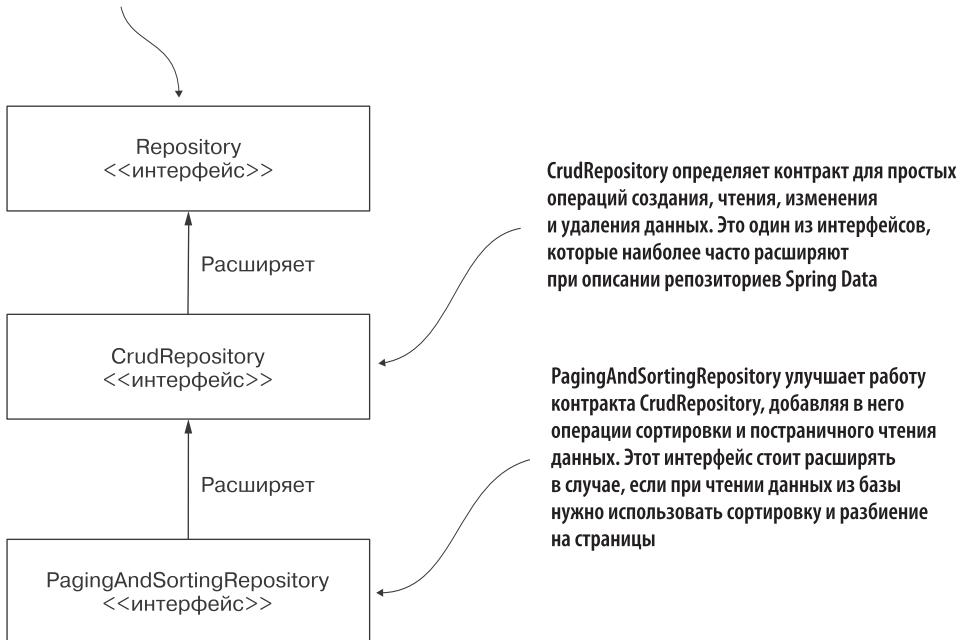


Рис. 14.6. Для реализации репозиториев приложения с помощью Spring Data нужно расширить определенный интерфейс. Основными интерфейсами, представляющими контракты Spring Data, являются Repository, CrudRepository и PagingAndSortingRepository. Для подключения функций приложения по взаимодействию с сохраненными данными нужно расширить один из этих контрактов

ПРИМЕЧАНИЕ

Не путайте аннотацию @Repository, описанную в главе 4, с интерфейсом Repository из проекта Spring Data. @Repository — это стереотипная аннотация, применяемая к классам, экземпляры которых Spring должен добавить в контекст приложения. Интерфейс Repository, о котором идет речь в этой главе, относится только к Spring Data, и, как вы узнаете, чтобы определить репозиторий Spring Data, нужно расширить этот интерфейс или же другой интерфейс, расширяющий интерфейс Repository.

Возможно, вам интересно: почему Spring Data предоставляет несколько интерфейсов, расширяющих один другой? Почему бы не сделать один интерфейс,

включив в него все эти операции? Благодаря подобному подходу (который еще называют *принципом изоляции интерфейсов*), когда вместо одного «большого» контракта в Spring Data реализовано несколько контрактов, расширяющих друг друга, можно описать в приложении только необходимые операции. Например, если в приложении нужны только операции CRUD, достаточно расширить контракт `CrudRepository`. Операции сортировки и постраничного чтения записей не будут доступны, благодаря чему приложение получится более простым (рис. 14.7).

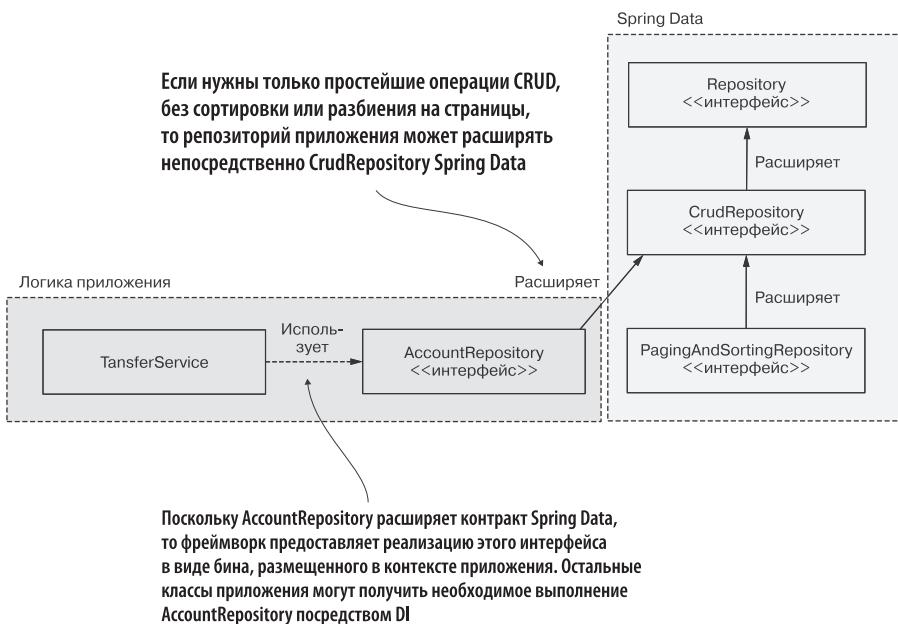


Рис. 14.7. Чтобы создать репозиторий Spring Data, нужно определить интерфейс, расширяющий один из контрактов Spring Data. Например, если в приложении понадобятся только операции CRUD, интерфейс, определенный как репозиторий, должен быть расширением `CrudRepository`. В контекст Spring-приложения добавляется бин, который реализует выбранный контракт, так что остальным заинтересованным компонентам приложения достаточно внедрить этот бин из контекста

Если в приложении, помимо операций CRUD, также нужны функции сортировки и постраничного чтения данных, следует расширить более специализированный интерфейс `PagingAndSortingRepository` (рис. 14.8).

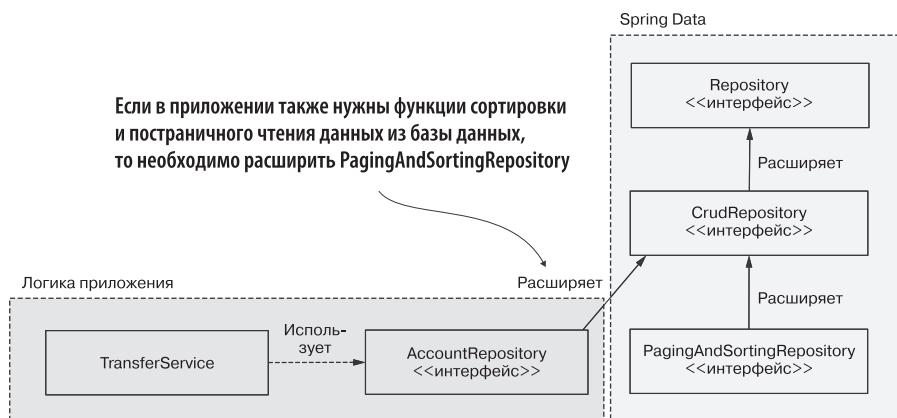


Рис. 14.8. Если в приложении нужны функции сортировки и постраничного чтения, необходимо расширить более специализированный контракт. В приложении создается бин, реализующий этот контракт. Затем бин может быть внедрен во все остальные компоненты приложения, которые должны его использовать

В некоторых модулях Spring Data могут быть и другие контракты, в зависимости от технологии, которую они представляют. Например, с помощью Spring Data JPA можно непосредственно расширить интерфейс `JpaRepository` (как показано на рис. 14.9). `JpaRepository` — еще более специализированный контракт, чем `PagingAndSortingRepository`. Он добавляет операции, применимые только при использовании определенных технологий, реализующих спецификацию Jakarta Persistence API (JPA), таких как Hibernate.

Другой пример — реализация технологии NoSQL, такой как MongoDB. Чтобы использовать Spring Data для MongoDB, нужно подключить к приложению модуль Spring Data Mongo. Этот модуль предоставляет, в частности, специализированный контракт `MongoRepository`, который добавляет в приложение операции, свойственные данной технологии хранения информации.

Если в приложении используется определенная технология, то в нем обычно расширяются те контракты Spring Data, в которых объявлены операции, свойственные именно этой технологии. Если в приложении не требуется ничего сверх операций CRUD, можно по-прежнему ограничиться расширением `CrudRepository`, но специализированные контракты, как правило, предоставляют более удобные решения, созданные для конкретной технологии. На рис. 14.10 показан класс `AccountRepository` (класс приложения), который расширяет `JpaRepository` (специальный модуль Spring Data JPA).

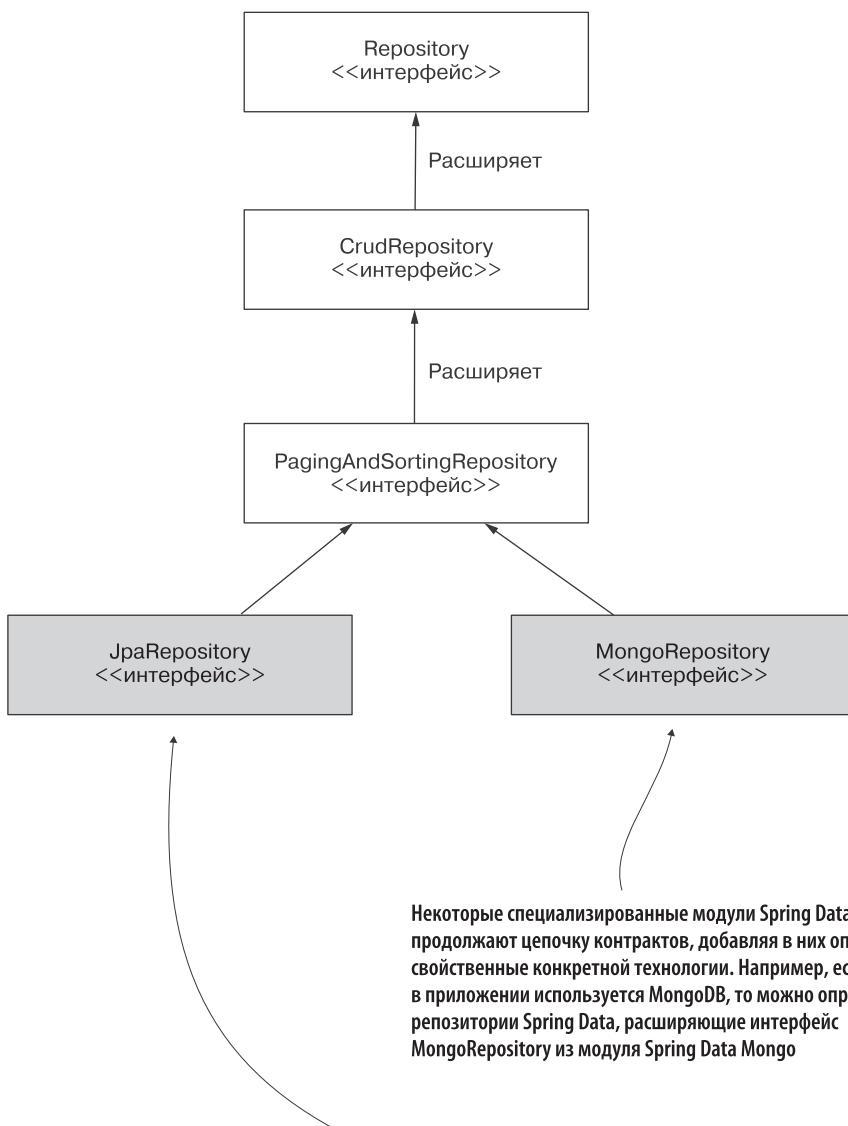


Рис. 14.9. В модулях Spring Data, соответствующих данным технологиям, обычно содержатся специализированные контракты, определяющие операции, которые можно использовать только в рамках этих технологий. Если в приложении есть технологии, то в нем, скорее всего, будут и необходимые контракты

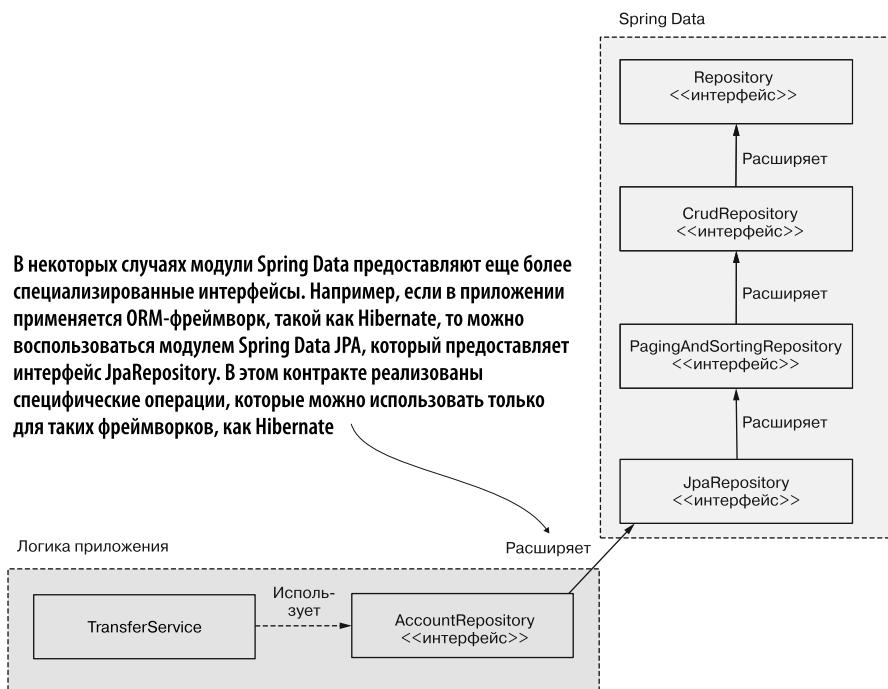


Рис. 14.10. В Spring Data есть и другие модули, которые дают еще более специализированные контракты. Например, при использовании ORM-фреймворка (такого как Hibernate, где реализована спецификация JPA) в сочетании со Spring Data можно расширить интерфейс JpaRepository — еще более специализированный контракт, предоставляющий операции,ственные только таким JPA-реализациям, как Hibernate

14.3. ИСПОЛЬЗОВАНИЕ SPRING DATA JDBC

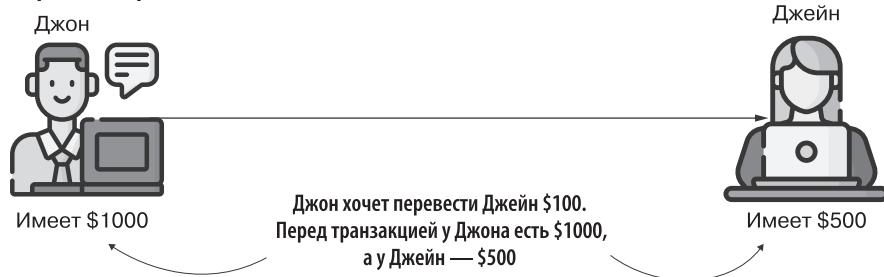
Для реализации уровня хранения данных в Spring-приложении мы воспользуемся модулем Spring Data JDBC. Как уже было сказано, нужно только расширить контракт Spring Data. Посмотрим, как это работает. Вы научитесь не только внедрять простой репозиторий, но также создавать и использовать дополнительные операции взаимодействия с ним.

Мы рассмотрим задачу, подобную той, над которой работали в главе 13, — создадим приложение, представляющее собой электронный кошелек для управления счетами пользователей. Пользователь может перевести деньги с одного счета на другой. В этом примере мы разработаем сценарий использования «перевод

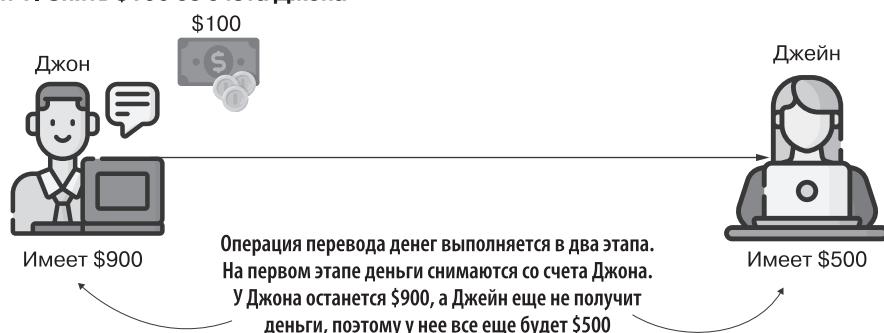
денег», который позволит отправлять средства с одного счета на другой. Операция перевода состоит из двух частей (рис. 14.11).

1. Снять заданную сумму со счета отправителя.
2. Положить эту сумму на счет получателя.

До операции перевода денег



Этап 1. Снять \$100 со счета Джона



Этап 2. Положить \$100 на счет Джейн

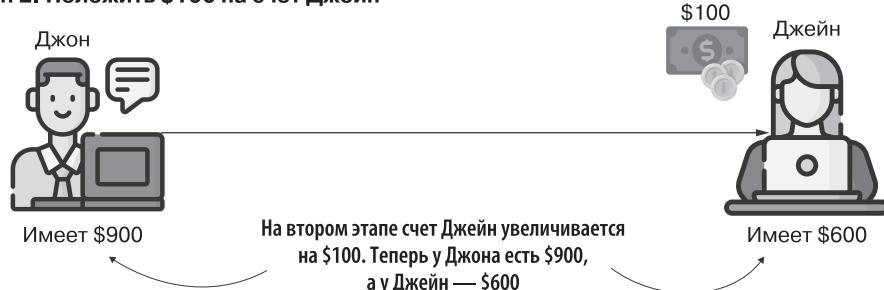


Рис. 14.11. Сценарий «перевод денег» выполняется в два этапа. Сначала приложение снимает переводимую сумму со счета отправителя (Джона). Затем приложение кладет переводимую сумму на счет получателя (Джейн)

Мы будем держать информацию о счетах в таблице базы данных. Чтобы пример был как можно более простым и коротким и вы могли сосредоточиться на теме этого раздела, мы будем использовать базу данных H2, хранящуюся в оперативной памяти (о которой уже говорилось в главе 12).

Таблица счетов будет состоять из следующих полей:

- `id` — первичный ключ, автоматически увеличивающееся поле типа INT;
- `name` — имя владельца счета;
- `amount` — сумма денег, которую владелец хранит на счету.

Данный пример находится в проекте sq-ch14-ex1. К проекту (в файле `pom.xml`) необходимо подключить следующие зависимости:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId> ← Для реализации уровня хранения
    <artifactId>spring-boot-starter-data-jdbc</artifactId> данных в приложении используем
</dependency>                                         модуль Spring Data JDBC
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

Мы также добавим в папку ресурсов проекта Maven файл `schema.sql`, чтобы создать таблицу счетов в базе данных H2, хранящейся в оперативной памяти. В этом файле содержится следующий DDL-запрос, необходимый для создания таблицы счетов:

```
create table account (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(50) NOT NULL,
    amount DOUBLE NOT NULL
);
```

Нам также нужно добавить пару записей в таблицу счетов. Впоследствии, когда мы закончим разработку приложения, эти записи понадобятся для его тестирования. Создадим в папке ресурсов Maven файл `data.sql`. Чтобы добавить две записи в таблицу счетов, мы внесем в `data.sql` два оператора `INSERT`:

```
INSERT INTO account VALUES (NULL, 'Jane Down', 1000);
INSERT INTO account VALUES (NULL, 'John Read', 1000);
```

В конце раздела мы продемонстрируем работу приложения, переведя Джону от Джейн \$100. Построим модель записи из таблицы счетов в виде класса `Account`. Для представления каждого столбца таблицы в этом классе мы создадим поле соответствующего типа.

Напомню, что во избежание возможных проблем с точностью при выполнении арифметических операций для дробных чисел я советую использовать не `double` или `float`, а `BigDecimal`. Для некоторых операций Spring Data, таких как чтение данных из базы, Spring Data должен знать, какое поле соответствует первичному ключу таблицы. Чтобы обозначить первичный ключ, мы будем применять аннотацию `@Id`, как показано в листинге 14.1, представляющем класс модели `Account`.

Листинг 14.1. Класс `Account`, представляющий модель записи из таблицы счетов

```
public class Account {

    @Id ← Атрибут, моделирующий первичный ключ, отмечаем аннотацией @Id
    private long id;

    private String name;
    private BigDecimal amount;

    // Геттеры и сеттеры
}
```

Теперь, построив класс модели, можно разработать репозиторий Spring Data (листинг 14.2). Для этого приложения нам нужны только операции CRUD, поэтому мы напишем интерфейс, который расширяет `CrudRepository`. Во всех интерфейсах Spring Data содержатся следующие два типа, которые необходимо реализовать:

- класс модели (которую иногда называют сущностью), для которого мы пишем репозиторий;
- тип поля первичного ключа.

Листинг 14.2. Определение репозитория Spring Data

```
public interface AccountRepository
    extends CrudRepository<Account, Long> { ←
}

```

Первое значение, принадлежащее к параметризованному типу, — это тип класса модели, представляющего таблицу. Второй — тип поля первичного ключа

В результате расширения интерфейса `CrudRepository` нам станут доступны простейшие операции Spring Data, такие как получение значения по его первичному ключу, чтение всех записей из таблицы, удаление записей и т. п. Но это далеко не все, что можно реализовать посредством SQL-запросов. В реальном

приложении нам также понадобились бы дополнительные операции подобного плана. Как создать такую операцию для репозитория Spring Data?

В Spring Data это делается очень просто — иногда даже не приходится писать SQL-запрос. Проект интерпретирует имена методов на основе ряда правил для создания таких имен и генерирует запросы. Предположим, вы хотите написать операцию получения данных для всех счетов по заданному имени. В Spring Data для этого достаточно метода с именем `findAccountsByName`. Если имя метода начинается с `find`, Spring Data знает, что вы хотите прочитать что-то из базы с помощью запроса `SELECT`. Следующее слово, `Accounts`, сообщает, что именно вы хотите выбрать. Spring Data даже догадается, что вы имели в виду, назвав метод `findByName`. Он поймет ваше стремление, поскольку этот метод принадлежит репозиторию `AccountRepository`. Но в нашем примере я бы хотел быть максимально точным и сделать имена операций как можно более понятными. После `By` в имени метода Spring Data ожидает получить условие запроса (`WHERE`). В данном случае мы хотим выбрать записи по имени, то есть `ByName`, что Spring Data переведет как `WHERE name = ?`.

Преобразование имени метода в запрос, автоматически выполняемое Spring Data, визуально представлено на рис. 14.12.

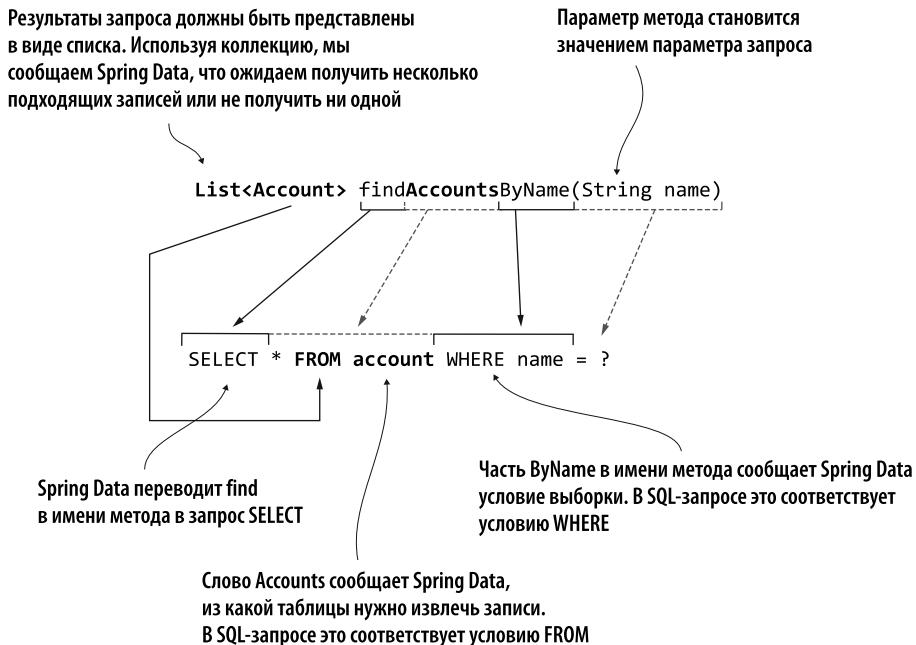


Рис. 14.12. Взаимосвязь между именем метода репозитория и запросом, генерируемым Spring Data

В листинге 14.3 показано определение метода в интерфейсе `AccountRepository`.

Листинг 14.3. Операция репозитория для получения данных обо всех счетах, открытых на заданное имя

```
public interface AccountRepository
    extends CrudRepository<Account, Long> {

    List<Account> findAccountsByName(String name);
}
```

На первый взгляд магия перевода имени метода в запрос кажется невероятной. Но со временем вы поймете, что это не панацея. У данного подхода есть ряд недостатков, и я всегда советую разработчикам определять запрос явно, вместо того чтобы рассчитывать на Spring Data. Главные недостатки состоят в следующем:

- если операция требует более сложного запроса, имя метода получится слишком длинным и его будет трудно читать;
- если при рефакторинге имя метода по ошибке изменят, нарушится поведение приложения, а вы этого можете даже не заметить (к сожалению, не все приложения тщательно тестируются — учитывайте данный факт);
- если только вы не работаете в IDE, где среда сама подсказывает правильное имя метода, правила именования методов в Spring Data придется выучить. Поскольку вы уже знакомы с SQL, то ничего не выигрываете, изучив еще и набор правил, применимый только для Spring Data;
- необходимость перевести имя метода в запрос снижает производительность, из-за чего замедляется инициализация приложения (так как процесс перевода происходит при запуске приложения).

Самый простой способ избежать этих проблем состоит в применении аннотации `@Query`. Она определяет SQL-запрос, который должно выполнить приложение при вызове метода. Если перед методом стоит `@Query`, то уже неважно, как вы его назовете. Spring Data не станет переводить имя метода в запрос, а возьмет тот запрос, который предоставите вы. Такое поведение также является более эффективным для производительности. Пример использования аннотации `@Query` показан в листинге 14.4.

Листинг 14.4. Использование аннотации `@Query` для определения SQL-запроса, необходимого для данной операции

```
public interface AccountRepository
    extends CrudRepository<Account, Long> {

    @Query("SELECT * FROM account WHERE name = :name")
    List<Account> findAccountsByName(String name);
}
```

Учтите, что имя параметра в запросе должно совпадать с именем параметра в методе. Между двоеточием (:) и именем параметра не должно быть пробела

Аннотация `@Query` используется одинаково для любых запросов. Но, если в запросе содержатся данные, необходимо также снабдить метод аннотацией `@Modifying`. `@Modifying` нужен и при запросах `UPDATE`, `INSERT` или `DELETE`. В листинге 14.5 показано, как использовать аннотацию `@Query` для создания метода репозитория с запросом `UPDATE`.

Листинг 14.5. Определение изменяющей операции в репозитории

```
public interface AccountRepository
    extends CrudRepository<Account, Long> {
    @Query("SELECT * FROM account WHERE name = :name")
    List<Account> findAccountsByName(String name);
    @Modifying ←
    @Query("UPDATE account SET amount = :amount WHERE id = :id")
    void changeAmount(long id, BigDecimal amount);
}
```

Перед методами, которые определяют операции, изменяющие данные, ставится аннотация `@Modifying`

Чтобы получить бин, реализующий интерфейс `AccountRepository`, можно воспользоваться DI. Не стоит беспокоиться из-за того, что вы написали только интерфейс. Spring Data сам создаст динамическую реализацию и добавит бин в контекст приложения. В листинге 14.6 показано, как компонент приложения `TransferService` получает бин `AccountRepository` посредством внедрения в конструктор. Как вы помните из главы 5, при запросе DI для поля интерфейсного типа Spring понимает, что нужно найти бин, который реализует этот интерфейс.

Листинг 14.6. Внедрение репозитория в класс сервиса для реализации сценария использования

```
@Service
public class TransferService {
    private final AccountRepository accountRepository;
    public TransferService(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }
}
```

В листинге 14.7 показана реализация сценария использования «перевод денег». С помощью `AccountRepository` мы получаем данные о счете и изменяем его баланс. Мы по-прежнему применяем аннотацию `@Transactional`, чтобы, как говорилось в главе 13, логика выполнялась в рамках транзакции и данные бы не нарушились в случае, если одна из операций завершится неудачно.

Листинг 14.7. Реализация сценария использования «перевод денег»

```

@Service
public class TransferService {

    private final AccountRepository accountRepository;

    public TransferService(AccountRepository accountRepository) {
        this.accountRepository = accountRepository;
    }

    @Transactional
    public void transferMoney(
        long idSender,
        long idReceiver,
        BigDecimal amount) {

        Account sender =
            accountRepository.findById(idSender)
                .orElseThrow(() -> new AccountNotFoundException());
    }

    Account receiver =
        accountRepository.findById(idReceiver)
            .orElseThrow(() -> new AccountNotFoundException());

    BigDecimal senderNewAmount =
        sender.getAmount().subtract(amount);
    BigDecimal receiverNewAmount =
        receiver.getAmount().add(amount);

    accountRepository
        .changeAmount(idSender, senderNewAmount);
    accountRepository
        .changeAmount(idReceiver, receiverNewAmount);
}
}

```

Помещаем логику сценария использования в рамки транзакции — во избежание рассогласования данных в том случае, если одна из операций завершится неудачно

Получаем информацию о счетах отправителя и получателя

Вычисляем новые балансы этих счетов, снимая переводимую сумму со счета отправителя и зачисляя ее на счет получателя

Изменяем балансы счетов в базе данных

В сценарии использования «перевод денег» мы применили простой класс исключения при выполнении, который называется `AccountNotFoundException`. Определение этого класса выглядит так:

```
public class AccountNotFoundException extends RuntimeException { }
```

Создадим метод сервиса, который будет читать записи из базы данных и получать все сведения о счетах для заданного владельца. Эти операции нам понадобятся при тестировании приложения. Чтобы получить все записи, нам не нужно писать метод вручную. Интерфейс `AccountRepository` наследует метод `findAll()` от контракта `CrudRepository`, как показано в листинге 14.8.

Листинг 14.8. Создание методов сервисов для получения информации о счетах

```

@Service
public class TransferService {

    // Остальной код

    public Iterable<Account> getAllAccounts() {
        return accountRepository.findAll(); ←
    }

    public List<Account> findAccountsByName(String name) {
        return accountRepository.findAccountsByName(name);
    }
}

```

AccountRepository наследует этот метод от интерфейса CrudRepository, принадлежащего Spring Data

В листинге 14.9 показано, как AccountController предоставляет доступ к сценарию использования «перевод денег» через конечную точку REST.

Листинг 14.9. Предоставление доступа к сценарию использования «перевод денег» посредством конечной точки REST

```

@RestController
public class AccountController {

    private final TransferService transferService;

    public AccountController(TransferService transferService) {
        this.transferService = transferService;
    }

    @PostMapping("/transfer")
    public void transferMoney( ←
        @RequestBody TransferRequest request
    ) {
        transferService.transferMoney( ←
            request.getSenderId(),
            request.getReceiverId(),
            request.getAmount()); ←
    }
}

```

Извлекаем из тела HTTP-запроса идентификаторы счетов отправителя и получателя, а также сумму перевода

Вызываем сервис для выполнения сценария использования «перевод денег»

В следующем фрагменте кода показана реализация DTO TransferRequest, который используется в конечной точке /transfer для преобразования тела HTTP-запроса:

```

public class TransferRequest {

    private long senderAccountId;
    private long receiverAccountId;
    private BigDecimal amount;

    // Геттеры и сеттеры
}

```

В листинге 14.10 создается конечная точка для чтения записей из базы данных.

Листинг 14.10. Конечная точка для получения информации о счете

```
@RestController
public class AccountController {

    // Остальной код

    @GetMapping("/accounts")
    public Iterable<Account> getAllAccounts( ←
        @RequestParam(required = false) String name ←
    ) {
        if (name == null) { ←
            return transferService.getAllAccounts(); ←
        } else { ←
            return transferService.findAccountsByName(name); ←
        }
    }
}
```

Используем дополнительный параметр запроса, чтобы передать имя владельца, информацию о счете которого нужно получить

Если мы не передадим имя в виде дополнительного параметра, вернется информация обо всех счетах

Если среди параметров запроса есть имя, мы получим только информацию о счете пользователя с этим именем

Запустим приложение и проверим записи, вызвав конечную точку `/accounts`, которая возвращает все счета из базы данных:

```
curl http://localhost:8080/accounts
```

Выполнив эту команду, вы увидите в консоли примерно следующее:

```
[{"id":1,"name":"Jane Down","amount":1000.0}, {"id":2,"name":"John Read","amount":1000.0}]
```

Чтобы перевести \$100 Джону от Джейн, вызовем конечную точку `/transfer`, используя команду cURL:

```
curl -XPOST -H "content-type:application/json" -d '{"senderAccountId":1, "receiverAccountId":2, "amount":100}' http://localhost:8080/transfer
```

Если снова вызовем конечную точку `/accounts`, увидим, что суммы на счетах изменились. После выполнения перевода у Джейн осталось всего \$900, а у Джона теперь есть \$1100:

```
curl http://localhost:8080/accounts
```

Результат вызова конечной точки `/accounts` после перевода денег должен выглядеть так:

```
[ {"id":1,"name":"Jane Down","amount":900.0}, {"id":2,"name":"John Read","amount":1100.0} ]
```

Если при вызове конечной точки `/accounts` использовать параметр запроса `name`, сформируется другой запрос, который позволит увидеть только счета Джейн:

```
curl http://localhost:8080/accounts?name=Jane+Down
```

Как показано в следующем фрагменте, в теле ответа, полученного в результате выполнения этой команды, присутствует информация, касающаяся только Джейн:

```
[  
  {  
    "id": 1,  
    "name": "Jane Down",  
    "amount": 900.0  
  }  
]
```

РЕЗЮМЕ

- Spring Data — это проект экосистемы Spring, который упрощает создание уровня хранения данных в Spring-приложениях. Spring Data предоставляет уровень абстракции, который объединяет многочисленные технологии хранения данных и помогает реализовать данную функцию, предоставляя единый набор контрактов.
- Благодаря Spring Data можно создавать репозитории посредством интерфейсов, которые расширяют следующие стандартные контракты:
 - `Repository`, не предоставляющий операции с сохраненными данными;
 - `CrudRepository`, который дает только простейшие операции CRUD — `CREATE`, `READ`, `UPDATE` и `DELETE`;
 - `PagingAndSortingRepository`, расширяющий `CrudRepository`, добавляя операции сортировки и постраничного чтения найденных записей.
- При использовании Spring Data необходимо выбрать определенный модуль в зависимости от применяемой в приложении технологии хранения данных. Например, если приложение соединяется с СУБД посредством JDBC, нужно подключить Spring Data JDBC, а в случае NoSQL, например MongoDB, понадобится Spring Data Mongo.
- При расширении контракта Spring Data приложение наследует и может использовать операции, определенные в этом контракте. Но можно также создать и дополнительные операции — в виде методов, определенных в интерфейсах репозитория.
- Отметив аннотацией `@Query` метод, объявленный в репозитории Spring Data, можно определить SQL-запрос, который будет выполняться для данной операции.

- Если объявить метод без явного указания запроса с помощью аннотации `@Query`, Spring Data преобразует имя этого метода в SQL-запрос. Чтобы имя метода можно было правильно расшифровать и корректно перевести в запрос, оно должно быть построено в соответствии с правилами Spring Data. Если Spring Data не сможет преобразовать имя метода в SQL-запрос, приложение не будет запущено и выбросит исключение.
- Лучше не полагаться на то, что Spring Data сам преобразует имя метода в запрос, а использовать аннотацию `@Query`. В случае перевода имен возможны следующие проблемы:
 - для более сложных операций имена методов получаются слишком длинными и трудночитаемыми, что усложняет поддержку приложения;
 - преобразование имен замедляет инициализацию приложения;
 - вам придется выучить соглашения об именовании методов Spring Data;
 - есть риск нарушить поведение приложения в результате некорректного рефакторинга с изменением имени метода.
- Любая операция, изменяющая данные (с выполнением запроса `INSERT`, `UPDATE` или `DELETE`) должна сопровождаться аннотацией `@Modifying`, которая показывает Spring Data, что данная операция модифицирует записи, хранящиеся в базе данных.

15

Тестирование Spring-приложений

В этой главе

- ✓ Почему так важно тестировать приложения.
- ✓ Как работают тесты.
- ✓ Как разрабатываются модульные тесты для Spring-приложений.
- ✓ Как разрабатываются интеграционные тесты для Spring.

В этой главе вы научитесь разрабатывать тесты для Spring-приложений. Тест – это небольшой фрагмент логики, назначение которого – убедиться, что определенная функция приложения работает так, как должна. Тесты делятся на две категории:

- *модульные тесты* – касаются только определенного участка логики;
- *интеграционные тесты* – предназначены для проверки правильного взаимодействия нескольких компонентов.

Но, говоря о тестах, я имею в виду обе эти категории.

Тесты необходимы для любого приложения. Они гарантируют, что изменения, которые вносятся в процессе разработки приложения, не нарушают уже существующие функции (или как минимум снижают вероятность появления ошибок). Кроме того, тесты играют роль документации. Многие разработчики, к сожалению, ими пренебрегают, поскольку они не являются непосредственной частью бизнес-логики приложения, но при этом, разумеется, на их написание уходит

какое-то время. Из-за этого кажется, что тесты не особенно и важны. Однако в действительности, хоть их значение не всегда заметно с первого взгляда, поговорите, в долговременной перспективе тесты неоценимы. Я не устаю подчеркивать, насколько важно следить, чтобы логика приложения всегда была как следует протестирована.

Зачем писать тесты, если можно просто отладить функцию вручную и успокоиться? Потому что:

- тесты можно запускать снова и снова, постоянно таким образом проверять адекватность функционирования приложения и, затрачивая минимум усилий, убеждаться, что все работает правильно;
- тесты можно использовать в качестве документации: читая этапы тестирования, мы легко понимаем назначение конкретного сценария использования;
- если в процессе разработки в приложении возникнут проблемы, благодаря тестам вы быстро об этом узнаете.

Почему приложение вдруг перестает работать, если раньше все было хорошо? Потому что мы постоянно модифицируем его исходный код, исправляем ошибки и добавляем новые опции. Внося изменения, можно нарушить реализованный ранее функционал.

Если написать тесты для этого функционала, можно выполнять их всякий раз, когда в приложении происходят изменения, чтобы проверить адекватность его работы. Если какая-нибудь из уже настроенных опций будет нарушена, вы узнаете об этом прежде, чем код будет сдан в эксплуатацию. *Регрессивное тестирование* — это постоянное тестирование существующего функционала с целью убедиться, что все работает правильно.

Хорошим подходом является создать тесты для всех сценариев, соответствующих каждой разработанной функции. Затем их можно выполнять всякий раз, когда вы что-нибудь измените, для уверености, что изменения не повлияли на созданный ранее функционал.

В настоящее время нельзя полагаться только на то, что разработчики выполняют все тесты вручную — тестирование становится частью процесса создания приложения. Как правило, в группах разработчиков практикуют то, что принято называть принципом *непрерывной интеграции* (continuous integration, CI): берется инструмент наподобие Jenkins или TeamCity и настраивается так, чтобы выполнять сборку всякий раз, когда в приложение вносится изменение. Инструмент непрерывной интеграции — это программный продукт, применяемый для проведения операций, который необходим для сборки и — иногда — установки приложений в процессе разработки. Инструмент CI также выполняет тесты и отправляет уведомления разработчикам, если что-то ломается (рис. 15.1).

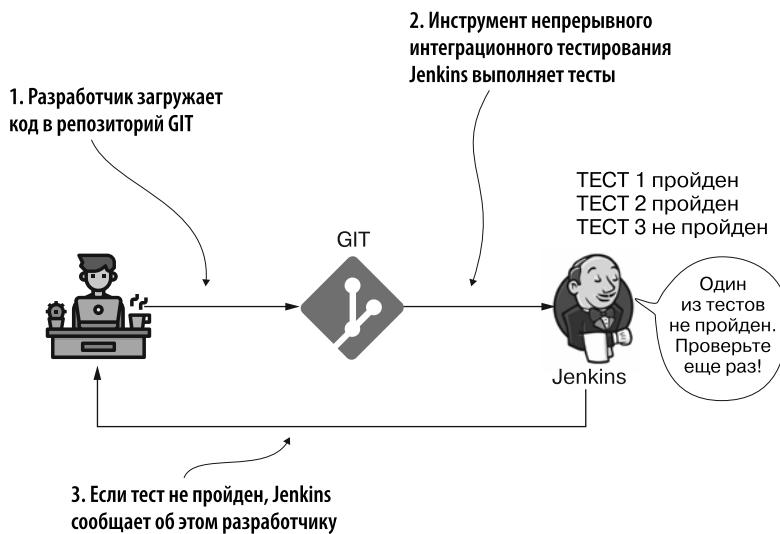


Рис. 15.1. Инструмент CI, например Jenkins или TeamCity, выполняет тесты всякий раз, когда разработчик вносит изменения в приложение. Если один из тестов не проходит, инструмент сообщает, что нужно проверить правильность работы функции и исправить проблему

В начале раздела 15.1 мы сформируем общее представление о том, что такое модульный тест и как он работает. В разделе 15.2 мы рассмотрим два наиболее часто встречающихся типа тестов, которые вы будете использовать для Spring-приложений: модульные и интеграционные. В качестве примеров мы напишем тесты для функций, разработанных ранее в этой книге.

Прежде чем углубиться в предмет, я бы хотел предупредить вас, что тестирование — сложная тема, и мы рассмотрим только самые важные вещи, которые необходимо знать при проверке Spring-приложений. Однако данная сфера за-служивает целой книжной полки. Советую вам изучить книгу Cătălin Tudose *JUnit in Action* (Manning, 2020), где вы найдете гораздо больше ценной информации о тестировании.

15.1. КАК ПИСАТЬ ПРАВИЛЬНЫЕ ТЕСТЫ

Обсудим, как функционируют тесты и что собой представляет правильно написанный тест. Вы научитесь разрабатывать код приложения таким образом, чтобы его было удобно тестировать, и увидите, что существует прямая связь между приложением, которое удобно тестировать, и приложением, которое легко поддерживать (то есть легко вносить изменения для реализации новых

функций и устранения ошибок). Тестируемость и поддерживаемость — характеристики, усиливающие друг друга. Проектируя тестируемое приложение, вы также делаете его поддерживаемым.

Тесты пишут для уверенности, что логика определенного метода, реализованного в проекте, работает так, как предполагалось. При тестировании конкретного метода обычно требуется проверить несколько сценариев (вариантов поведения приложения в зависимости от разных входных условий). Для каждого такого сценария пишется тестовый метод в тестовом классе. В проектах Maven (подобных тем, которые мы создавали для примеров ранее) тестовые классы размещаются в папке `test` (рис. 15.2).

Тестовый класс относится только к тому методу, логика которого проверяется. Даже простая логика требует нескольких сценариев. В тестовом классе нужно для каждого написать метод, который будет проверять этот сценарий.

Рассмотрим пример. Помните сценарий использования «перевод денег» в главах 13 и 14? Это была простая реализация отправки заданной суммы с одного счета на другой. Данный сценарий использования состоял всего из нескольких операций.

1. Найти в базе данные о счете отправителя.
2. Найти в базе данные о счете получателя.
3. Вычислить новые балансы для обоих счетов после перевода.
4. Изменить значения балансов в базе данных.

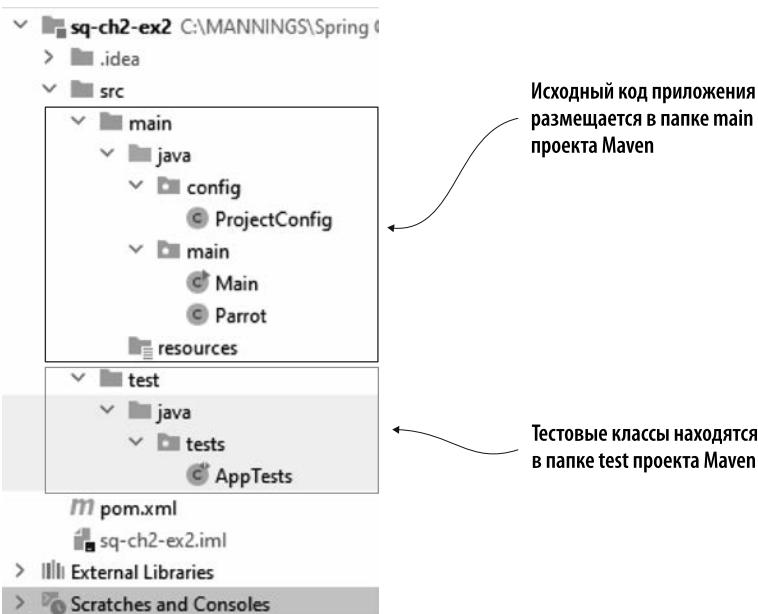


Рис. 15.2. В проекте Maven тестовые классы размещаются в папке `test`

Даже для этих операций есть несколько сценариев, которые требуют тестирования.

1. Что произойдет, если приложение не найдет данных о счете отправителя.
2. Что произойдет, если приложение не найдет данных о счете получателя.
3. Что произойдет, если на счету отправителя окажется недостаточно денег.
4. Что произойдет, если не удастся изменить балансы счетов.
5. Что произойдет, если все операции будут успешно выполнены.

Для каждого сценария тестирования нужно понять, как должно вести себя приложение, и написать тестовый метод, позволяющий убедиться, что все работает соответствующим образом. Например, если в тестовом случае 3 необходимо отменить перевод, когда на счету отправителя недостаточно денег, то при тестировании приложение должно выбрасывать определенное исключение и операция не должна выполняться. Однако в зависимости от требований, предъявляемых к приложению, можно ввести кредитный лимит для счета отправителя. В таком случае тест должен учитывать также и это.

Реализация тестовых сценариев тесно связана с тем, как должно работать конкретное приложение. Но с технической точки зрения идея всегда одна и та же: нужно определить сценарии и написать тестовый метод для каждого из них (рис. 15.3).

В проекте Maven исходный код приложения размещается в папке main

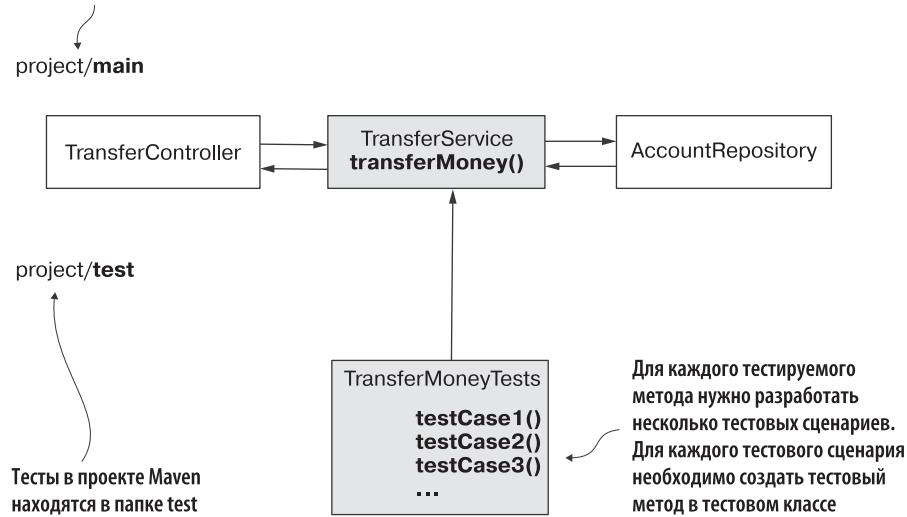


Рис. 15.3. Для любой тестируемой логики нужно сформулировать все тестовые сценарии. Для каждого нужно написать тестовый метод в тестовом классе. Тестовые классы размещаются в папке `test` проекта Maven. На этом рисунке `TransferMoneyTests` — это тестовый класс, в котором содержатся тестовые сценарии для метода `transferMoney()`. В `TransferMoneyTests` определено несколько методов тестирования для всех релевантных сценариев логики, реализованной в `transferMoney()`

Критически важный момент, на который необходимо обратить внимание, — даже для маленького метода существует несколько возможных тестовых сценариев — и это еще одна причина, почему методы в приложении должны быть как можно короче! Если писать длинные методы с большим количеством строк кода и многими параметрами — выполняющие сразу несколько действий, — то составить список релевантных сценариев для них становится невероятно сложно. Считается, что если создавать для каждой обязанности отдельный небольшой и легкий для чтения метод, тестируемость приложения возрастает.

15.2. РЕАЛИЗАЦИЯ ТЕСТОВ В SPRING-ПРИЛОЖЕНИЯХ

Далее мы используем две технологии тестирования Spring-приложений, которые будут вам часто встречаться в реальных проектах. Каждую из них мы опробуем на примере сценария использования, знакомого нам из предыдущих глав, для которого мы будем писать тесты. По моему мнению, данные технологии обязательно должен знать любой разработчик:

- *написание модульных тестов для проверки логики метода.* Модульные тесты короткие, быстро выполняются и касаются только одного процесса. Они позволяют сосредоточиться на проверке маленького фрагмента логики, исключив остальные зависимости;
- *написание интеграционных тестов Spring для проверки логики метода в сочетании со специальными возможностями, предоставляемыми фреймворком.* Эти тесты позволяют убедиться, что функции приложения будут правильно работать при обновлении зависимостей.

В подразделе 15.2.1 вы познакомитесь с модульными тестами. Мы обсудим, почему они так важны, и рассмотрим этапы их создания. В качестве примера вы также напишете несколько модульных тестов для сценариев использования, реализованных в предыдущих главах. В подразделе 15.2.2 вы научитесь разрабатывать интеграционные тесты. Вы узнаете, чем они отличаются от модульных тестов и как дополняют модульные тесты в Spring-приложении.

15.2.1. Разработка модульных тестов

Ниже мы изучим модульные тесты. Модульные тесты — это методы, которые вызывают определенный сценарий использования, чтобы проверить его поведение в специфических условиях. В модульном тестовом методе определяются условия, для которых выполняется сценарий использования, и проверяется, соответствует ли поведение метода требованиям приложения. В модульных тестах исключаются все зависимости тестируемой функции и рассматривается только данный изолированный фрагмент логики.

Ценность модульных тестов состоит в том, что если один из них выполняется с ошибкой, то вы знаете, с какой частью кода связана проблема, — тест показывает, где именно нужно внести исправления. Модульные тесты подобны лампочкам на приборной панели автомобиля. Если вы заводите машину, а она не заводится, причиной может быть как отсутствие бензина, так и поломка аккумулятора. Автомобиль — сложная система (как и приложение), и только индикаторы способны подсказать, в чем проблема. Если они показывают, что закончился бензин, вы легко поймете, что случилось!

Цель модульных тестов — проверить поведение отдельной части логики. Подобно индикаторам на приборной панели автомобиля, они помогают идентифицировать проблемы, связанные с определенными компонентами.

Создание первого модульного теста

Рассмотрим один из сценариев использования, который мы разработали в главе 14: перевод денег. В этом фрагменте логики выполняются следующие операции.

1. Найти данные о счете отправителя.
2. Найти данные о счете получателя.
3. Вычислить новый баланс для каждого счета.
4. Изменить баланс счета отправителя.
5. Изменить баланс счета получателя.

Реализация сценария использования, который мы разработали в проекте sq-ch14-ex1, представлена в листинге 15.1.

Листинг 15.1. Реализация сценария использования «перевод денег»

```
@Transactional
public void transferMoney(
    long idSender,
    long idReceiver,
    BigDecimal amount) {
    Account sender = accountRepository.findById(idSender)
        .orElseThrow(() -> new AccountNotFoundException());
    Account receiver = accountRepository.findById(idReceiver)
        .orElseThrow(() -> new AccountNotFoundException());

    BigDecimal senderNewAmount = sender.getAmount().subtract(amount);
    BigDecimal receiverNewAmount = receiver.getAmount().add(amount);

    accountRepository
        .changeAmount(idSender, senderNewAmount);
    accountRepository
        .changeAmount(idReceiver, receiverNewAmount);
}
```

Как правило, самый очевидный сценарий, для которого пишут первый тест, — это сценарий успешного выполнения, где не возникает исключений или ошибок. Успешное выполнение сценария использования «перевод денег» представлено на рис. 15.4.

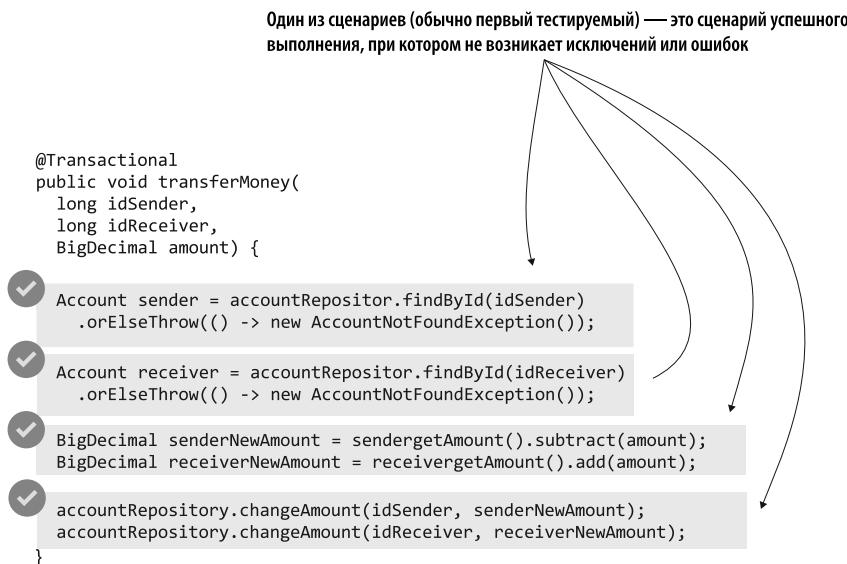


Рис. 15.4. Сценарий успешного выполнения, при котором не возникает ошибок и исключений. Как правило, вначале пишут тест именно для такого сценария, поскольку он наиболее очевиден

Напишем модульный тест для сценария успешного выполнения перевода денег. Любой тест состоит из трех основных частей (рис. 15.5).

1. *Предпосылки.* Определить входные данные и найти все зависимости для проверяемой логики, чтобы получить желаемый сценарий выполнения. Какие входные данные следует предоставить и как должны вести себя зависимости тестируемой логики, чтобы она работала так, как мы хотим?
2. *Вызов/выполнение.* Вызвать тестируемую логику и проверить ее поведение.
3. *Проверки.* Определить все операции проверки, которые нужно выполнить для данной части логики. Что должно произойти при вызове данной логики для заданных условий?

ПРИМЕЧАНИЕ

Иногда эти три этапа (предпосылки, вызов, проверки) называют иначе: Arrange, Act, and Assert (настроить, выполнить, подтвердить) или Given, When, and Then (дано, если, то). Но, как бы их ни называли, концепция написания тестов остается прежней.

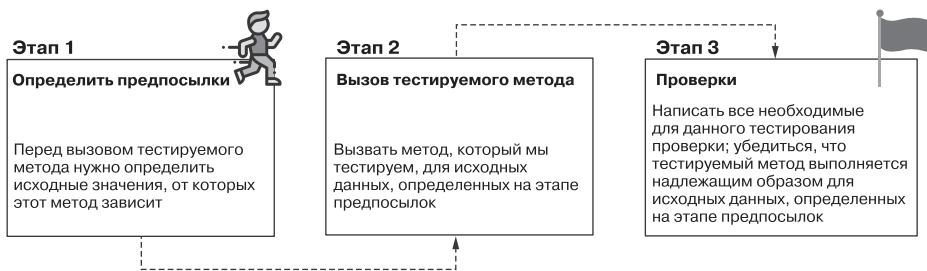


Рис. 15.5. Этапы написания модульного теста: сформулируйте исходные условия, определив данные для метода; вызовите метод; напишите проверки, которые должны выполниться в teste, чтобы убедиться в правильном поведении метода

В *предпосылках* нужно понять, какие зависимости задействованы в том тестовом случае, для которого мы пишем тест. Здесь выбираются такие входные данные и такое поведение зависимостей, чтобы тестируемая логика вела себя определенным образом.

Какие зависимости нужны для сценария использования «перевод денег»? Зависимости — это все, что используется в методе, но не создается непосредственно в нем:

- параметры метода;
- экземпляры объектов, которые используются в методе, но не создаются в нем.

Зависимости для нашего примера показаны на рис. 15.6.

Параметры — это зависимости выполнения.
Их значения определяют поведение метода

```

@Transactional
public void transferMoney(
    long idSender,
    long idReceive,
    BigDecimal amount) {
    Account sender = accountRepository.findById(idSender)
        .orElseThrow(() -> new AccountNotFoundException());
    Account receiver = accountRepository.findById(idReceiver)
        .orElseThrow(() -> new AccountNotFoundException());
    BigDecimal senderNewAmount = sender.getAmount().subtract(amount);
    BigDecimal receiverNewAmount = receiver.getAmount().add(amount);
    accountRepository.changeAmount(idSender, senderNewAmount);
    accountRepository.changeAmount(idReceive, receiverNewAmount);
}

```

Другие объекты, которые являются внешними по отношению к методу, но используются в методе для реализации логики, — это тоже зависимости выполнения. Их поведение определяет и поведение метода

Рис. 15.6. Модульный тест проверяет логику сценария использования без учета каких-либо зависимостей. Чтобы написать подобный тест, нужно знать обо всех зависимостях и уметь ими управлять. В нашем случае такими зависимостями являются параметры метода и объект AccountRepository

Запуская метод для тестирования, мы можем передать любые значения всех трех параметров, чтобы управлять процессом выполнения. Но с экземпляром `AccountRepository` ситуация немного сложнее. Выполнение метода `transferMoney()` зависит от поведения метода `findById()`, принадлежащего `AccountRepository`.

Следует учитывать, что модульный тест касается только одного фрагмента логики приложения, поэтому он не должен вызывать метод `findById()`. Модульный тест должен предполагать, что `findById()` работает так, как нужно, и что при выполнении тестируемого метода происходит то, что и должно происходить в данной ситуации.

Однако тестируемый метод вызывает метод `findById()`. Как с этим быть? Чтобы контролировать такие зависимости, используют *заглушки* (*mocks*) — псевдообъекты, поведением которых можно управлять. В данном случае мы сделаем так, чтобы вместо настоящего объекта `AccountRepository` в тестируемом методе использовался псевдообъект. Мы воспользуемся возможностью повлиять на поведение этого псевдообъекта, чтобы вызвать все возможные варианты выполнения метода `transferMoney()`, которые хотим протестировать.

На рис. 15.7 показано, что именно мы собираемся сделать: заменить объект `AccountRepository` на заглушку и таким образом убрать зависимость из тестируемого объекта.

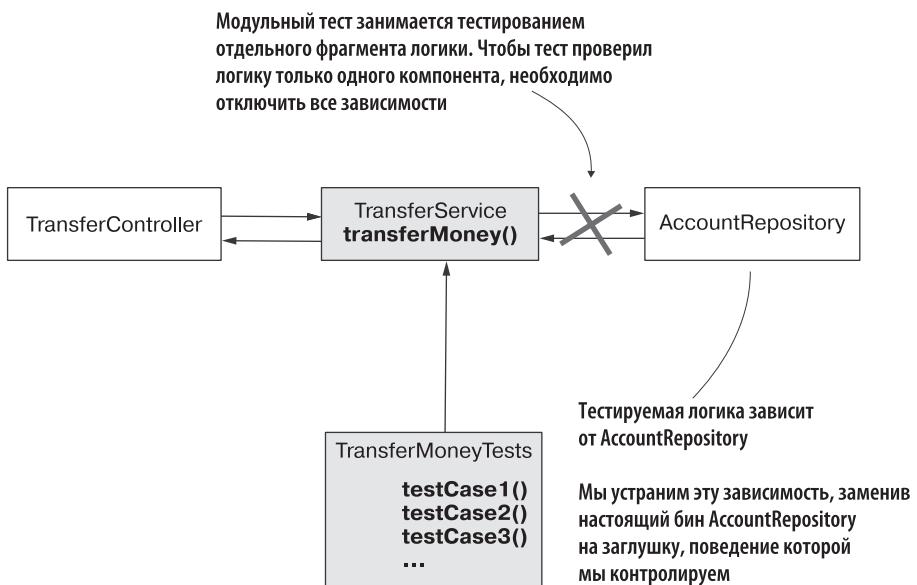


Рис. 15.7. Чтобы модульный тест касался только логики метода `transferMoney()`, мы исключили зависимость от `AccountRepository`. Мы заменили настоящий объект `AccountRepository` на заглушку и управляем этим псевдообъектом, чтобы протестировать поведение `transferMoney()` в разных ситуациях

В листинге 15.2 мы начнем разработку модульного теста. Создав новый класс в папке `test` проекта Maven, создадим первый тестовый сценарий, написав новый метод с аннотацией `@Test`.

ПРИМЕЧАНИЕ

Для создания модульных и интеграционных тестов в примерах, рассматриваемых в этой книге, мы будем использовать JUnit 5 Jupiter — последнюю версию JUnit. Но на практике вам также будет часто встречаться JUnit 4. Это еще одна причина, по которой я советую почитать специализированные книги о тестировании. Глава 4 книги Cătălin Tudose *JUnit in Action* (Manning, 2020) посвящена различиям между JUnit 4 и JUnit 5.

Для вызова тестируемого метода `transferMoney()` мы создадим экземпляр `TransferService`. Вместо настоящего экземпляра `AccountRepository` мы возьмем объект-заглушку, которым сможем управлять. Такие заглушки создаются с помощью метода `mock()`. Метод `mock()` становится доступен благодаря зависимости `Mockito` (часто используемой для создания тестов с помощью JUnit).

Листинг 15.2. Создание объекта, метод которого нужно протестировать посредством модульных тестов

```
public class TransferServiceUnitTests {
    @Test
    public void moneyTransferHappyFlow() {
        AccountRepository accountRepository = ←
            mock(AccountRepository.class); ←
        TransferService transferService = ←
            new TransferService(accountRepository);
    }
}
```

С помощью метода Mockito `mock()` создаем экземпляр-заглушку для объекта `AccountRepository`

Создаем экземпляр объекта `TransferService`, метод которого хотим протестировать. Вместо настоящего экземпляра `AccountRepository` создаем объект, играющий роль его заглушки. Таким образом мы заменим зависимость на объект, которым можем управлять

Теперь мы можем задать поведение объекта-заглушки, после чего вызвать тестируемый метод и убедиться, что он правильно работает в созданных условиях. Как показано в листинге 15.3, для управления действиями заглушки используется метод `given()`. С помощью `given()` можно определить, как будет вести себя заглушки при вызове одного из ее методов. В данном случае нам нужно, чтобы метод `findById()` объекта `AccountRepository` возвращал определенный экземпляр `Account` в зависимости от заданного значения параметра.

ПРИМЕЧАНИЕ

Как вы увидите в следующем листинге, в реальных проектах хорошим тоном считается описывать сценарии тестирования с помощью аннотации `@DisplayName`. В наших примерах я опустил `@DisplayName`, чтобы сэкономить место и чтобы вы могли сосредоточиться на логике выполнения теста. Но в реальных приложениях использование аннотации `@DisplayName` поможет лучше понять сценарий тестирования — не только вам, но и другим участникам процесса разработки.

Листинг 15.3. Модульный тест, проверяющий вариант успешного выполнения метода

```
public class TransferServiceUnitTests {
    @Test
    @DisplayName("Test the amount is transferred " +
        "from one account to another if no exception occurs.")
    public void moneyTransferHappyFlow() {
        AccountRepository accountRepository =
            mock(AccountRepository.class);
        TransferService transferService =
            new TransferService(accountRepository);

        Account sender = new Account(); ←
        sender.setId(1);
        sender.setAmount(new BigDecimal(1000)); ←

        Account destination = new Account(); ←
        destination.setId(2);
        destination.setAmount(new BigDecimal(1000)); ←

        given(accountRepository.findById(sender.getId()))
            .willReturn(Optional.of(sender));

        given(accountRepository.findById(destination.getId())) ←
            .willReturn(Optional.of(destination));

        transferService.transferMoney( ←
            sender.getId(),
            destination.getId(),
            new BigDecimal(100) ←
        );
    }
}
```

Управляемый нами метод `findById()`, получая ID счета отправителя, возвращает экземпляр этого счета. Данную строку следует читать так: «Если вызвать `findById()` и передать ему ID счета отправителя в виде параметра, этот метод вернет экземпляр счета отправителя»

Создаем экземпляры `Account` для отправителя и получателя, где хранится информация об их счетах — предполагается, что в действительности приложение получает ее из базы данных

Управляемый нами метод `findById()` получает ID счета получателя и возвращает экземпляр этого счета. Данную строку следует читать так: «Если вызвать `findById()` и передать ему ID счета получателя в виде параметра, этот метод вернет экземпляр счета получателя»

Вызываем метод, который хотим протестировать, передавая ему ID отправителя, ID получателя и сумму перевода

Последнее, что нам осталось сделать, — сообщить тесту, что должно произойти при выполнении метода. Чего мы ожидаем? Мы знаем, что назначение метода состоит в переводе денег с одного заданного счета на другой. Таким образом, мы ожидаем, что он вызовет экземпляр репозитория и присвоит балансам соответствующие значения. В листинге 15.4 добавлены тестовые инструкции, которые проверяют, правильно ли метод вызывает методы экземпляра репозитория, чтобы изменить балансы счетов.

Чтобы проверить, вызывается ли метод объекта-заглушки, как показано в листинге 15.4, используется метод `verify()`.

Листинг 15.4. Модульный тест, проверяющий успешное выполнение метода

```
public class TransferServiceUnitTests {

    @Test
    public void moneyTransferHappyFlow() {
        AccountRepository accountRepository =
            mock(AccountRepository.class);
        TransferService transferService =
            new TransferService(accountRepository);

        Account sender = new Account();
        sender.setId(1);
        sender.setAmount(new BigDecimal(1000));

        Account destination = new Account();
        destination.setId(2);
        destination.setAmount(new BigDecimal(1000));

        given(accountRepository.findById(sender.getId()))
            .willReturn(Optional.of(sender));

        given(accountRepository.findById(destination.getId()))
            .willReturn(Optional.of(destination));

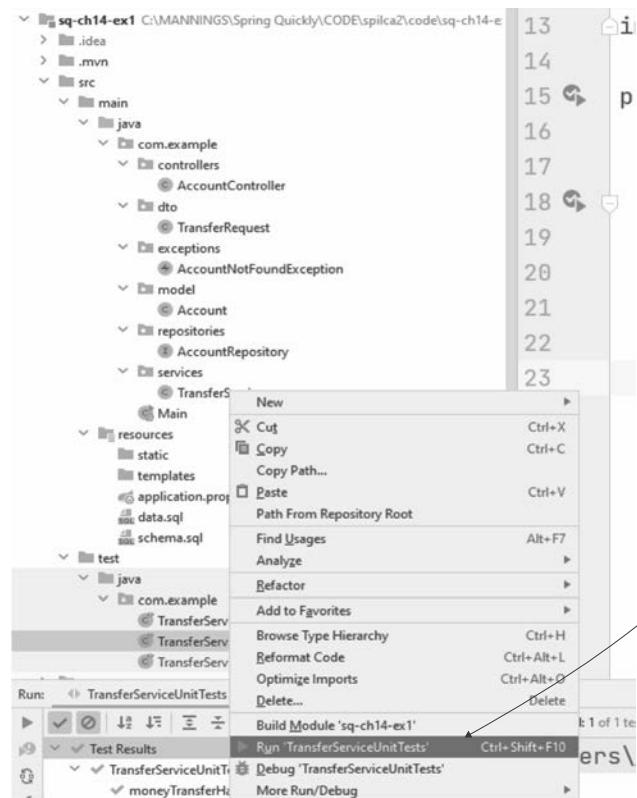
        transferService.transferMoney(
            sender.getId(),
            destination.getId(),
            new BigDecimal(100)
        );

        verify(accountRepository)
            .changeAmount(1, new BigDecimal(900));
        verify(accountRepository)
            .changeAmount(2, new BigDecimal(1100));
    }
}
```

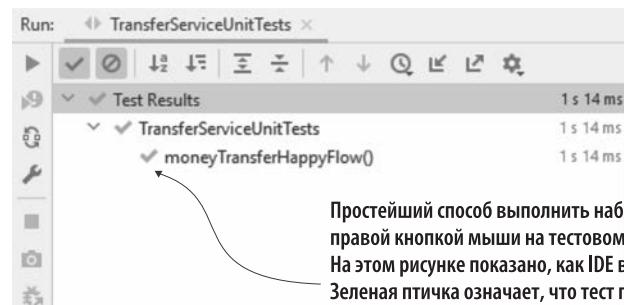
Проверяем, что метод changeAmount()
из AccountRepository вызван
с ожидаемыми параметрами

Если сейчас выполнить тест (обычно для этого нужно щелкнуть правой кнопкой мыши на тестовом классе в IDE и выбрать команду Run tests), результат его прохождения будет успешным. В этом случае данный тест в IDE отмечается зеленым цветом, а в консоли не появляется сообщений об исключениях. Если же тест не пройден, он обычно подсвечивается красным или желтым (рис. 15.8).

Вам будет часто встречаться такое объявление метода `mock()` внутри тестового метода, как показано в листингах 15.2–15.4. Однако я обычно предпочитаю другой способ создания объекта-заглушки. Нельзя сказать, чтобы он всегда был лучше или охотнее применялся, но я считаю, что код будет понятнее, если описывать заглушку и тестируемый объект с помощью аннотаций, как в листинге 15.5.



Простейший способ выполнить набор тестов в любой IDE — щелкнуть правой кнопкой мыши на тестовом классе и выбрать команду Run



Простейший способ выполнить набор тестов в любой IDE — щелкнуть правой кнопкой мыши на тестовом классе и выбрать команду Run. На этом рисунке показано, как IDE выводит результаты тестиования. Зеленая птичка означает, что тест пройден успешно. Красный или желтый крестик показывает, что тест не пройден — при этом в консоли появится сообщение с объяснением, почему именно это произошло

Рис. 15.8. Выполнение теста. Обычно в IDE можно провести тест несколькими способами. Один из них — щелкнуть правой кнопкой мыши на тестовом классе и выбрать команду Run. Или же можно выполнить все тесты проекта, нажав правой кнопкой мыши на имя проекта и выбрав Run tests. Графические интерфейсы разных IDE могут немного различаться, но все они выглядят приблизительно так, как показано на рисунке. После проведения тестов в IDE отображаются результаты каждого из них

Листинг 15.5. Создание заглушек для зависимостей с помощью аннотаций

```

@ExtendWith(MockitoExtension.class) ← С помощью аннотации @Mock
public class TransferServiceWithAnnotationsUnitTests { ← создаем объект-заглушку и внедряем
    private AccountRepository accountRepository; ← его в поле тестового класса,
                                                 перед которым стоит эта аннотация

    @Mock ← Разрешаем использование
    private TransferService transferService; ← аннотаций @Mock и @InjectMocks

    @Test
    public void moneyTransferHappyFlow() {
        Account sender = new Account();
        sender.setId(1);
        sender.setAmount(new BigDecimal(1000));

        Account destination = new Account();
        destination.setId(2);
        destination.setAmount(new BigDecimal(1000));

        given(accountRepository.findById(sender.getId()))
            .willReturn(Optional.of(sender));

        given(accountRepository.findById(destination.getId()))
            .willReturn(Optional.of(destination));

        transferService.transferMoney(1, 2, new BigDecimal(100));
    }

    verify(accountRepository)
        .changeAmount(1, new BigDecimal(900));

    verify(accountRepository)
        .changeAmount(2, new BigDecimal(1100));
    }
}

```

Обратите внимание: вместо того чтобы объявлять эти объекты внутри тестового метода, я вынес их за пределы метода и объявил атрибуты класса с аннотациями `@Mock` и `@InjectMocks`. Встретив аннотацию `@Mock`, фреймворк создает объект-заглушку и внедряет его в атрибут, перед которым стоит аннотация. В случае `@InjectMocks` фреймворк создает тестируемый объект и внедряет все заглушки (созданные с помощью `@Mock`) в его параметры.

Чтобы `@Mock` и `@InjectMocks` работали, нужно также поставить перед тестовым классом аннотацию `@ExtendWith(MockitoExtension.class)`. Таким образом мы активируем расширение, позволяющее фреймворку читать `@Mock` и `@InjectMocks` и управлять полями с этими аннотациями.

На рис. 15.9 показан результат работы нашего теста. На схеме видно, какие операции мы выполнили и какой код написали для выполнения каждой из них. Сравните их с теми, которые мы перечислили перед началом разработки теста.

1. *Предпосылки* — найти все зависимости и взять на себя управление ими.
2. *Выполнение* — выполнить тестируемый метод.
3. *Проверка* — проверить, соответствует ли поведение выполняемого метода нашим ожиданиям.

```
@ExtendWith(MockitoExtension.class)
public class TransferServiceWithAnnotationsUnitTests {

    ① @Mock
    private AccountRepository accountRepository;

    @InjectMocks
    private Transferservice transfer Service;

    @Test
    public void moneyTransferHappyFlow() {
        Account sender = new Account();
        sender.setIld(1);
        sender.setAmount(new BigDecimal(1000));

        Account destination = new Account();
        destination.setIld(2);
        destination.setAmount(new BigDecimal(1000));

        given (accountRepository.findById(sender.getId()))
            .willReturn(Optional.of(sender));

        given (accountRepository.findById(destination.getId()))
            .willReturn(Optional.of(destination));

        ② transferService.transferMoney(1,2, new BigDecimal(100));
        verify(accountRepository).changeAmount(1, new BigDedmal(900));
        verify(accountRepository).changeAmount(2, new BigDedmal(1100));
    }
}
```

Рис. 15.9. Основные этапы выполнения теста: 1 — определяем зависимости и берем на себя управление ими; 2 — выполняем тестируемый метод; 3 — проверяем, соответствует ли поведение метода ожидаемому

Создание тестов для выполнения с исключением

Напомню: в тестировании нуждается не только вариант успешного выполнения метода. Желательно также убедиться, что метод ведет себя ожидаемым образом при возникновении исключений. Такой вариант называется *выполнением*

с исключением. В нашем примере он может иметь место, если не будет найден счет отправителя или получателя по заданному ID (рис. 15.10).

Успешное выполнение не единственный исход событий, который должен покрываться тестами. Необходимо идентифицировать все возможные для данного сценария использования варианты и создать тесты для проверки поведения приложения. Например, что произойдет, если не будет найден счет получателя? Ожидается, что приложение выбросит соответствующее исключение

```
@Transactional
public void transferMoney(
    long idSender,
    long idReceiver,
    BigDecimal amount) {
    ✓ Account sender = accountRepository.findById(idSender)
        .orElseThrow(() -> new AccountNotFoundException());
    ✗ Account receiver = accountRepository.findById(idReceiver)
        .orElseThrow(() -> new AccountNotFoundException());
    BigDecimal senderNewAmount = sender.getAmount().subtract(amount);
    BigDecimal receiverNewAmount = receiver.getAmount().add(amount);
    accountRepository.changeAmount(idSender, senderNewAmount);
    accountRepository.changeAmount(idReceiver, receiverNewAmount);
}
```

В данном случае, если не удалось найти данные о счете получателя, вычисление новых балансов и обновление информации на счетах не должно выполняться

Рис. 15.10. Выполнение с исключением — это такое функционирование метода, при котором возникает ошибка или выбрасывается исключение. Например, если не найдена информация о счете получателя, приложение должно выдать исключение `AccountNotFoundException`, а метод `changeAmount()` не должен выполняться. Варианты выполнения с исключениями тоже важны, поэтому и для этих сценариев, как и для успешного выполнения, необходимо разработать тесты

В листинге 15.6 показано, как написать модульный тест для выполнения метода с исключением. Дабы убедиться, что тестируемый метод выбрасывает исключение, используем метод `assertThrows()`. Ему нужно сообщить, какой метод вам нужен и какое исключение тот должен выбросить. `assertThrows()` вызывает тестируемый метод и проверяет, реагирует ли он ожидаемым исключением.

Листинг 15.6. Тестирование выполнения с исключением

```

@ExtendWith(MockitoExtension.class)
public class TransferServiceWithAnnotationsUnitTests {

    @Mock
    private AccountRepository accountRepository;

    @InjectMocks
    private TransferService transferService;

    @Test
    public void moneyTransferDestinationAccountNotFoundFlow() {
        Account sender = new Account();

        sender.setId(1);
        sender.setAmount(new BigDecimal(1000));

        given(accountRepository.findById(1L))
            .willReturn(Optional.of(sender));

        given(accountRepository.findById(2L))
            .willReturn(Optional.empty()); ←

        assertThrows(
            AccountNotFoundException.class, ←
            () -> transferService.transferMoney(1, 2, new BigDecimal(100))
        );
    }

}

```

Управляемая заглушкой `AccountRepository`, мы делаем так, чтобы метод `findById()`, вызванный для счета получателя, возвращал пустой объект `Optional`

Мы предполагаем, что для данного варианта выполнения метод должен выбрасывать исключение `AccountNotFoundException`

Используем метод `verify()` с условием `never()` для уверенности, что метод `changeAmount()` не вызывается

Тестирование значения, возвращаемого методом

Часто бывает так, что нужно проверить, какое значение возвращает метод. В листинге 15.7 показан метод, разработанный нами в главе 9 в проекте `sq-ch9-ex1`. Как разработать модульный тест для этого метода в случае, когда пользователь предоставляет правильные данные для аутентификации?

Листинг 15.7. Действие контроллера аутентификации, которое мы хотим протестировать

```

@PostMapping("/")
public String loginPost(
    @RequestParam String username,
    @RequestParam String password,
    Model model
) {
    loginProcessor.setUsername(username);
    loginProcessor.setPassword(password);
    boolean loggedIn = loginProcessor.login();
}

```

```

if (loggedIn) {
    model.addAttribute("message", "You are now logged in.");
} else {
    model.addAttribute("message", "Login failed!");
}

return "login.html";
}

```

Нам нужно выполнить все те же операции, о которых уже говорилось ранее.

1. Найти зависимости и взять на себя управление ими.
2. Вызвать тестируемый метод.
3. Убедиться, что при выполнении поведение тестируемого метода соответствует ожидаемому.

Реализация модульного теста показана в листинге 15.8. Обратите внимание: мы заменили заглушками те зависимости, чьим поведением хотим управлять или чье поведение хотим проверить, — объекты `Model` и `LoginProcessor`. Мы сделали так, чтобы заглушка для `LoginProcessor` возвращала `true` (что соответствует ситуации, когда пользователь предоставил правильные данные для аутентификации). Теперь мы вызовем метод, который хотим протестировать.

Мы проверим следующее:

- что метод возвращает строку "login.html". Для проверки значения, выдаваемого методом, мы воспользуемся одним из методов семейства `assert` — как показано в листинге 15.8, это будет `assertEquals()`. Он сравнивает ожидаемое значение с тем значением, которое дает тестируемый метод;
- что экземпляр `Model` содержит правильное сообщение — "You are now logged in". С помощью метода `verify()` мы убедимся, что при вызове метода `addAttribute()` экземпляра `Model` этому методу в качестве параметра передается правильное значение.

Листинг 15.8. Тестирование возвращаемого значения с помощью модульного теста

```
@ExtendWith(MockitoExtension.class)
class LoginControllerUnitTests {
```

```
    @Mock
    private Model model;
```



Определяем объекты-заглушки и внедряем их в экземпляр, поведение которого хотим протестировать

```
    @Mock
    private LoginProcessor loginProcessor;
```



```
    @InjectMocks
    private LoginController loginController;
```



```

@Test
public void loginPostLoginSucceedsTest() {
    given(loginProcessor.login())
        .willReturn(true);
    String result = loginController.loginPost("username", "password", model);
    assertEquals("login.html", result);
    verify(model)
        .addAttribute("message", "You are now logged in.");
}
}

Делаем так, чтобы управляемый
нами объект-заглушка LoginProcessor
при вызове метода login() возвращал true

Вызываем тестируемый метод
созданными предпосылками

Проверяем значение, которое
возвращает тестируемый метод

Проверяем значение
атрибута сообщения,
который был добавлен
к объекту модели

```

Благодаря тому что мы контролируем входные данные (значения параметров и поведение объектов-заглушек), мы можем протестировать, что произойдет при разных вариантах выполнения метода. В листинге 15.9 мы сделали так, чтобы метод `login()` объекта-заглушки `LoginProcessor` возвращал `false` — теперь можно посмотреть, что будет при неудачной попытке аутентификации.

Листинг 15.9. Добавление теста для проверки неудачной аутентификации

```

@ExtendWith(MockitoExtension.class)
class LoginControllerUnitTests {

    // Остальной код

    @Test
    public void loginPostLoginFailsTest() {
        given(loginProcessor.login())
            .willReturn(false);

        String result =
            loginController.loginPost("username", "password", model);

        assertEquals("login.html", result);

        verify(model)
            .addAttribute("message", "Login failed!");
    }
}

```

15.2.2. Разработка интеграционных тестов

Интеграционные тесты очень похожи на модульные. Мы будем писать их с помощью уже знакомого вам JUnit. Но вместо того, чтобы сосредоточиться на работе отдельных компонентов, благодаря интеграционным тестам мы будем проверять взаимодействие двух и более компонентов.

Помните аналогию с индикаторами на приборной панели автомобиля? Если бак полон, но что-то сломалось в системе подачи бензина из бака в двигатель, машина не поедет. К сожалению, в этом случае индикатор не укажет на проблему: бензина достаточно и бензобак как изолированный компонент работает нормально. Так что мы не поймем, где истинная причина поломки. То же самое может случиться и в приложении. Компоненты могут работать правильно, будучи изолированными друг от друга, но некорректно «общаться» между собой. Разработка интеграционных тестов позволяет смягчить проблемы, случающиеся, если сами по себе компоненты работают адекватно, но совершают ошибки при взаимодействии.

Чтобы рассмотреть это на примере, нам понадобится то же приложение, которое мы использовали для модульных тестов, — сценарий использования «перевод денег», разработанный в главе 14 (проект `sq-ch14-ex1`).

Какой вид взаимодействий мы можем протестировать? Здесь возможны следующие варианты.

- *Взаимодействие между двумя (или более) объектами приложения.* Тестирование правильного взаимодействия объектов помогает обнаружить проблемы коммуникации, возникающие в том случае, если один из этих объектов изменяется.
- *Контакт между объектом приложения и функцией фреймворка, которая расширяет возможности этого объекта.* Проверка взаимодействия объекта с функциями помогает обнаружить проблемы обновления версии фреймворка. Интеграционный тест позволяет немедленно узнать, что во фреймворке что-то изменилось и функция, на основе которой создан данный объект, теперь работает иначе.
- *Интеграция приложения с уровнем хранения данных (базой данных).* Тестирование взаимодействия репозитория с базой данных гарантирует быстрое обнаружение проблем, появляющихся при обновлении или изменении зависимости, которая обеспечивает контакт приложения с сохраненными данными (такой как JDBC-драйвер).

На первый взгляд интеграционный тест очень похож на модульный. Он состоит из тех же этапов предпосылок, вызова тестируемого метода и проверки результатов. Однако теперь он не ограничивается изолированным фрагментом логики, и мы не обязаны заменять все зависимости заглушками. Мы можем позволить тестируемому методу вызывать метод, принадлежащий другому настоящему объекту (не заглушке), если хотим проверить правильное их взаимодействие. Поэтому, если в модульном teste необходимо заменять репозиторий заглушкой, то для интеграционного это не является обязательным требованием. Мы все еще можем использовать заглушку вместо репозитория, если пишем тест, в котором контакт сервиса с репозиторием не имеет значения. Но если мы хотим

протестировать взаимодействие этих двух объектов, необходимо вызывать реальные экземпляры (рис. 15.11).

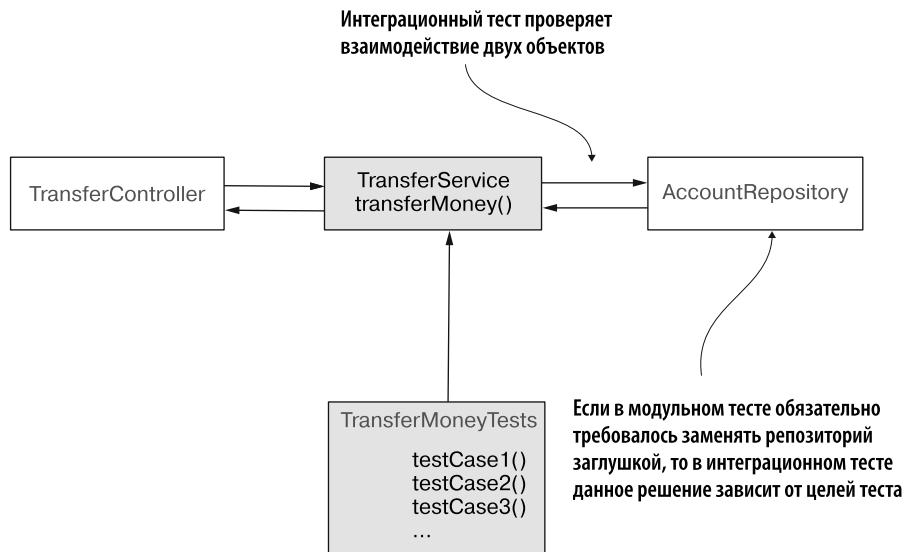


Рис. 15.11. В случае модульного теста все зависимости нужно было заменять заглушками. Если целью интеграционного теста является проверка взаимодействия между объектами TransferService и AccountRepository, в качестве репозитория нужно оставить реальный объект. В интеграционном teste по-прежнему можно заменять заглушками объекты, чье взаимодействие с тестируемым компонентом не проверяется

ПРИМЕЧАНИЕ

Если вы решите не заменять в интеграционном teste репозиторий заглушкой, вместо обычной базы данных используйте лучше базу, хранящуюся в оперативной памяти (например, H2), — так тест останется независимым от инфраструктуры, в рамках которой выполняется приложение. Настоящая база данных может привести к задержкам в работе тестов, а в случае проблем в инфраструктуре или сети — даже к неудачному их прохождению. Поскольку вы проверяете приложение, а не инфраструктуру, избегайте подобных случаев с помощью базы-заглушки.

В Spring-приложениях интеграционные тесты обычно используются для уверенности, что приложение правильно взаимодействует с функциями, предоставляемыми фреймворком. Такие тесты называют интеграционными тестами Spring. В отличие от модульных интеграционные тесты позволяют Spring создавать бины и изменять конфигурацию контекста (как это обычно происходит при запуске приложения).

В листинге 15.10 показано, как легко можно преобразовать модульный тест в интеграционный тест Spring. Обратите внимание на аннотацию `@MockBean`, с помощью которой создается объект-заглушка в нашем приложении Spring Boot. Эта аннотация очень похожа на аннотацию `@Mock`, применяемую в модульных тестах, но она также гарантирует, что объект-заглушка будет добавлен в контекст приложения. Таким образом, чтобы внедрить объект, поведение которого мы тестируем, достаточно воспользоваться аннотацией `@Autowired` (уже известной вам из главы 3).

Листинг 15.10. Интеграционный тест Spring

```
@SpringBootTest
class TransferServiceSpringIntegrationTests {

    @MockBean
    private AccountRepository accountRepository; // Создаем объект-заглушку,
                                                    // который входит в состав
                                                    // контекста Spring

    @Autowired
    private TransferService transferService; // Внедряем реальный объект
                                              // из контекста Spring, поведение
                                              // которого хотим протестировать

    @Test
    void transferServiceTransferAmountTest() {
        Account sender = new Account();
        sender.setId(1);
        sender.setAmount(new BigDecimal(1000));

        Account receiver = new Account();
        receiver.setId(2);
        receiver.setAmount(new BigDecimal(1000));

        when(accountRepository.findById(1L))
            .thenReturn(Optional.of(sender));
        when(accountRepository.findById(2L))
            .thenReturn(Optional.of(receiver));

        transferService
            .transferMoney(1, 2, new BigDecimal(100)); // Вызываем тестируемый метод

        verify(accountRepository)
            .changeAmount(1, new BigDecimal(900));
        verify(accountRepository)
            .changeAmount(2, new BigDecimal(1100));
    }
}
```

Создаем объект-заглушку, который входит в состав контекста Spring

Внедряем реальный объект из контекста Spring, поведение которого хотим протестировать

Определяем все предпосылки для теста

Проверяем, соответствует ли поведение тестируемого метода ожидаемому

ПРИМЕЧАНИЕ

Аннотация `@MockBean` – это аннотация Spring Boot. Если в вашем Spring-приложении нет Spring Boot, вы не сможете использовать `@MockBean`. Но вы сможете применить тот же подход, поставив перед классом конфигурации аннотацию `@ExtendsWith(SpringExtension.class)`. Пример кода с ней находится в проекте `sq-ch3-ex1`.

Данный тест выполняется так же, как и любой другой. Однако теперь Spring «видит» тестируемый объект и управляет им так же, как в рабочем приложении. Например, если после обновления версии фреймворка внедрение зависимости почему-либо перестанет работать, тест не будет пройден, несмотря на то что сам проверяемый объект не изменился. То же самое касается любой функции Spring, которая используется в тестируемом методе: транзакционности, безопасности, кеширования и т. п. Таким образом можно проверить интеграцию со всеми функциями, нужными методу в приложении.

ПРИМЕЧАНИЕ

На практике модульные тесты применяются для проверки поведения отдельных компонентов, а интеграционные тесты Spring — для необходимых сценариев интеграции. Несмотря на то что такие тесты тоже позволяют изучить реакцию компонента (если выполнить все тестовые сценарии для логики метода), использовать их таким образом — плохая идея. Интеграционные тесты занимают больше времени, поскольку затрагивают конфигурацию контекста Spring. Каждый вызов метода активирует несколько механизмов Spring, необходимых для работы фреймворка, — в зависимости от того, какие функции нужны данному методу. Нет смысла тратить время и ресурсы на реализацию каждого сценария тестирования логики приложения. Для экономии времени лучший вариант — проверять логику отдельных компонентов посредством модульных тестов и использовать интеграционные тесты только для уверенности, что метод взаимодействует с фреймворком правильно.

РЕЗЮМЕ

- Тест — это короткий фрагмент кода, который пишется для проверки поведения определенной логики, реализованной в приложении. Тесты необходимы, поскольку помогают гарантировать, что последующие усовершенствования приложения не нарушают уже существующих функций. Кроме того, тесты упрощают формирование документации.
- Тесты делятся на две категории: модульные и интеграционные. У каждого типа — свое назначение.
 - Модульные тесты касаются изолированного фрагмента логики. Они проверяют работу этого простого компонента, не затрагивая его интеграцию с другими функциями. Польза модульных тестов состоит в том, что они быстро выполняются и в случае проблемы в компоненте явно на нее указывают.
 - Интеграционные тесты предназначены для проверки взаимодействия между двумя и более компонентами. Такие тесты необходимы, поскольку два компонента могут правильно работать сами по себе, но некорректно взаимодействовать друг с другом. Интеграционные тесты помогают смягчить проблемы, возникающие в подобных случаях.

- Иногда при проверке необходимо устраниć зависимости от отдельных компонентов, чтобы тест затрагивал только определенные, а не все взаимодействующие части приложения. В таких случаях компоненты, которые не должны участвовать в тестировании, заменяются заглушками — псевдообъектами, находящимися под нашим контролем. Заглушки позволяют исключить нежелательные зависимости и сосредоточить тест только на конкретных контактах компонентов.
- Любой тест состоит из трех частей, таких как:
 - *предпосылки* — определение входных значений и поведения объектов-заглушек;
 - *вызов/выполнение* — вызов тестируемого метода;
 - *проверки* — проверка поведения тестируемого метода.

Приложение A

Архитектурные концепции

Ниже мы рассмотрим концепции архитектуры, с которыми вы можете встретиться. Для полного понимания информации, обсуждаемой в этой книге, вам необходимо хотя бы знать об их существовании и иметь общее представление о том, в чем они заключаются. В данном приложении мы пробежимся по монолитной и сервис-ориентированной архитектуре, а также архитектуре микросервисов. Я дам ссылки на другие ресурсы, с помощью которых вы сможете глубже изучить эти темы.

Архитектурные концепции — сложный предмет; ему посвящены многие книги и десятки презентаций, поэтому не могу обещать, что сделаю вас экспертом в вопросе, дав прочитать следующие несколько страниц. Тем не менее они помогут вам лучше понять, почему нужно использовать Spring в разных случаях, представленных в моей книге. В качестве примера возьмем сценарий работы приложения и рассмотрим историю изменения его архитектуры с первых лет создания приложений до настоящего времени.

A.1. МОНОЛИТНАЯ АРХИТЕКТУРА

Обсудим, что такое монолит. Вы поймете, почему раньше разработчики создавали монолитные приложения, — далее это поможет вам увидеть, как появились остальные архитектурные стили.

Когда разработчики называют приложение монолитным или монолитом, подразумевается, что оно состоит из единственного компонента. Именно этот компонент развертывается и выполняется, именно в нем реализован весь функционал продукта. В качестве примера рассмотрим приложение для управления книжным магазином. Пользователи могут управлять товарами, размещенными

на стеллажах, заказами, доставкой и данными о покупателях. Система, представленная на рис. А.1, является монолитной, поскольку все эти функции являются частью одного процесса.

ПРИМЕЧАНИЕ

То, что пользователь собирается делать с помощью приложения, называется бизнес-потоком. Например, когда владелец магазина продаёт книги, бизнес-поток заключается в следующем: функция управления товарами резервирует соответствующие книги на складе, функция формирования счетов выписывает счет на них, а функция доставки планирует, когда будет осуществляться их рассылка, и сообщает об этом покупателям. Диаграмма бизнес-потока продажи книг представлена на рис. А.2.

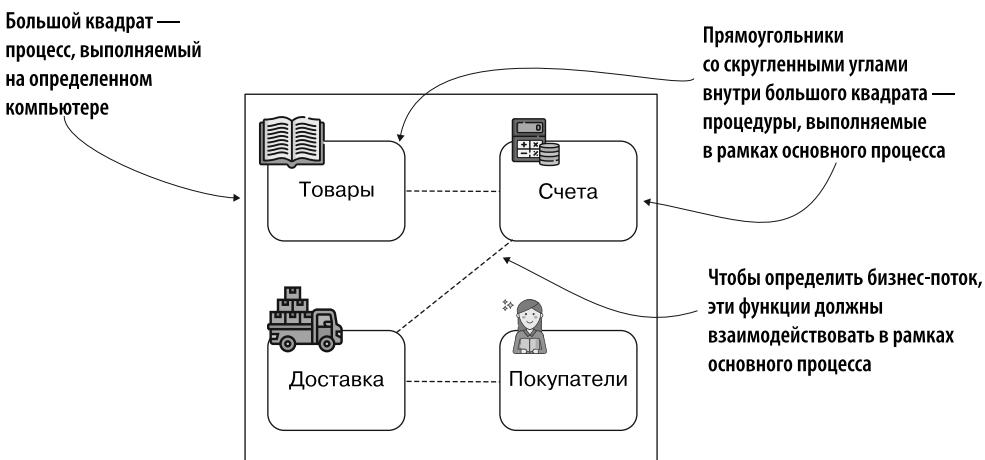


Рис. А.1. Монолитное приложение: весь функционал заключен в рамках одного процесса. Для реализации бизнес-потока функции взаимодействуют друг с другом внутри этого основного процесса

ПРИМЕЧАНИЕ

На рис. А.2 я показал упрощенную коммуникацию между компонентами, чтобы вы могли сосредоточиться на обсуждаемой теме. Структура классов, которая обеспечивает нужное взаимодействие, может отличаться от того, что вы видите на схеме.

Вначале все приложения создавались как монолитные, и на заре разработки такой подход прекрасно работал. В 1990-е годы интернет представлял собой сеть, состоящую всего из нескольких компьютеров; но прошло немного времени — и эта сеть разрослась до миллиарда устройств. Сегодня технологии перестали быть уделом технарей. Число пользователей и объемы данных, обрабатываемых многочисленными системами, значительно увеличились. А 30 лет назад возможность вызвать такси из любой точки, где бы вы ни оказались, или отправить сообщение, стоя на улице и ожидая возможности перейти дорогу, казалась чем-то за гранью воображения.

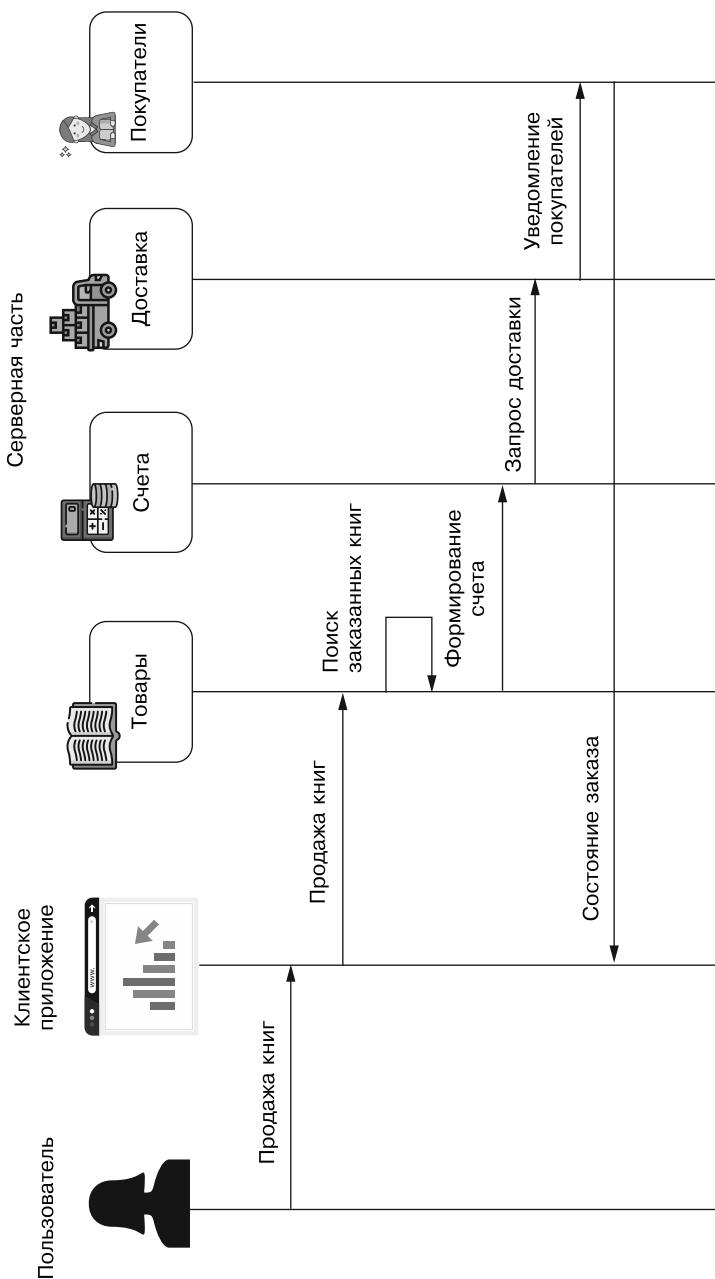


Рис. А.2. Пример бизнес-потока. Пользователь хочет продать книгу. Клиентское приложение отправляет запрос на сервер. У каждой обязанности сервера есть роль в общем потоке. Для выполнения бизнес-потока функции взаимодействуют между собой. В итоге клиентскому приложению сообщается статус заказа

Чтобы справиться с таким наплывом пользователей и данных, приложениям требуется больше ресурсов, и наличие всего лишь одного процесса усложняет управление этими ресурсами. К тому же за это время изменились не только количество пользователей и объемы данных — люди стали использовать приложения практически для всего, что только можно хотеть делать удаленно. Например, сегодня вы можете управлять своими банковскими счетами, попивая капучино в любимом кафе. Несмотря на видимую простоту процессов, они подразумевают повышенные риски для безопасности. Поэтому системы, обеспечивающие такие сервисы, должны быть хорошо защищенными и надежными.

Разумеется, все это также привело к модификации способов разработки приложений. Далее, чтобы было понятнее, учтем только увеличение количества пользователей. Что прежде всего стоит сделать, чтобы приложение смогло обслуживать больше запросов? Разумеется, запустить его на нескольких системах. Мы получим несколько экземпляров работающего приложения, запросы будут распределяться между ними, так что вся система сможет выдерживать более сильную нагрузку (рис. А.3). Этую концепцию называют *горизонтальным масштабированием*. Ради простоты будем считать, что рост происходит линейно: если один экземпляр работающего приложения способен обработать 50 000 запросов одновременно, то три экземпляра осилят до 150 000.

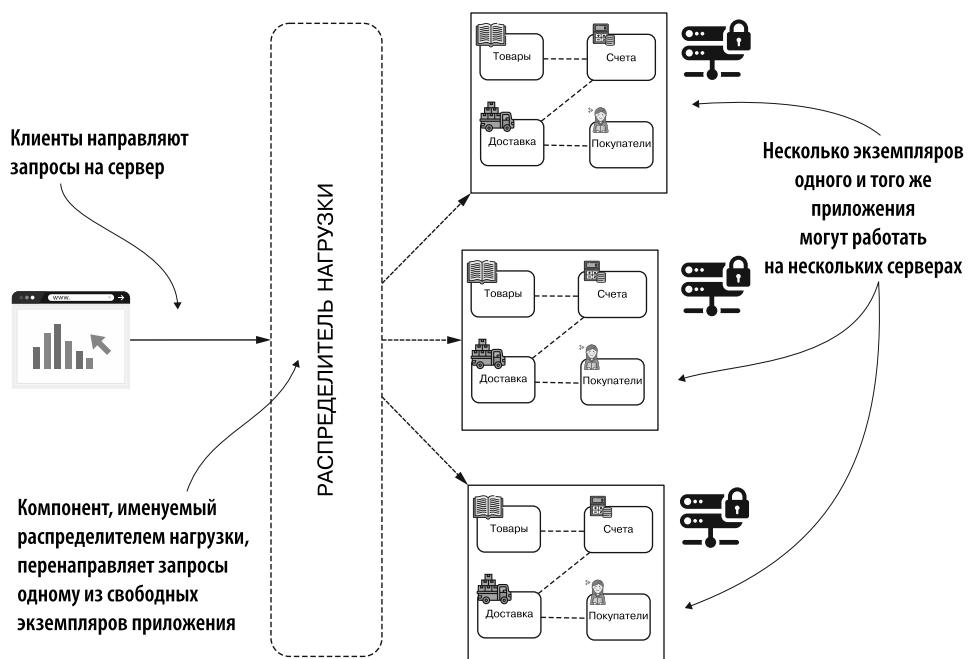


Рис. А.3. Горизонтальное масштабирование. Многократное выполнение одного и того же экземпляра позволяет задействовать больше ресурсов и обрабатывать больше клиентских запросов

Нужно учесть еще один аспект: приложения, как правило, постоянно развиваются. Даже незначительное изменение монолита требует заново развернуть все приложение, в то время как в архитектуре микросервисов понадобится только тот сервис, которого коснулись изменения. От подобного упрощения выигрывает вся система.

Почему бы не продолжать использовать монолитную архитектуру при разработке приложений? Возможно, в этом и нет проблемы. Подобно любой другой технологии или методике, есть задачи, для которых монолитная структура может быть наилучшим вариантом. Мы рассмотрим случаи, когда монолит не является правильным выбором, но я бы не хотел, чтобы у вас создалось впечатление, будто монолитная архитектура ни на что не годна, или что описанные мной концепции для разработки приложений априори подходят лучше всего.

Во многих случаях выбор монолитной архитектуры действительно является ошибочным. Мне часто приходится слышать жалобы разработчиков на то, что монолитные приложения сложно поддерживать. На самом деле проблема, как правило, заключается не в том, что приложение монолитное. Главная причина обычно кроется в запутанном коде или в том, что при разработке перепутали обязанности и неверно использовали абстракции. Однако монолитное приложение не всегда должно иметь запутанный код. Тем не менее по мере развития программного обеспечения стали возникать ситуации, в которых монолитная архитектура перестала работать и появилась потребность в альтернативах.

A.2. СЕРВИС-ОРИЕНТИРОВАННАЯ АРХИТЕКТУРА

Теперь можем перейти к сервис-ориентированной архитектуре. Используя пример из раздела A.1, мы покажем, что у монолитной концепции есть свои ограничения, поэтому в некоторых ситуациях нужен другой стиль разработки приложений. Вы узнаете, как сервис-ориентированная архитектура позволяет решить конкретные задачи, а также какие трудности она привносит в разработку приложений.

Вернемся к нашему приложению по продаже книг. В нем есть четыре функции: управление товарами, доставкой, счетами и покупателями. На практике не все функции требуют одинакового количества ресурсов. Одним нужно больше, чем другим, — как правило, потому, что какие-то функции сложнее или чаще используются.

В случае монолитной архитектуры нельзя масштабировать только часть приложения. Мы можем затронуть либо все четыре функции, либо ни одну из них.

Однако для более эффективного распределения ресурсов хотелось бы масштабировать только те функции, которым это действительно нужно (рис. А.4).



Рис. А.4. Некоторые функции используются интенсивнее, чем другие. Поэтому они потребляют больше ресурсов и нуждаются в масштабировании

Можно ли масштабировать только функцию управления товарами, не затрагивая остальные задачи? Да, для этого нужно разделить монолитную структуру на несколько сервисов. Мы изменим архитектуру приложения с монолитной на сервис-ориентированную (Service-Oriented Architecture, SOA). Вместо одного процесса для всех функций в SOA есть несколько процессов, где реализованы эти функции. Теперь мы можем масштабировать только тот из них, который имеет отношение к функции, требующей дополнительных ресурсов (рис. А.5).

Еще одно преимущество SOA — возможность изолировать обязанности: когда у вас есть отдельные приложения для управления счетами, доставкой и т. д., проще разделить их реализации и отслеживать взаимодействия между ними. В результате такую систему легче поддерживать. Еще одно следствие — легче управлять и командой разработчиков этой системы, так как вместо нескольких подразделений, работающих над одним приложением, можно создать группы, каждая из которых будет заниматься своим сервисом (рис. А.6).

В сервис-ориентированной архитектуре каждая функция представляет собой отдельный процесс. Благодаря этому можно масштабировать только те из них, которые потребляют больше ресурсов

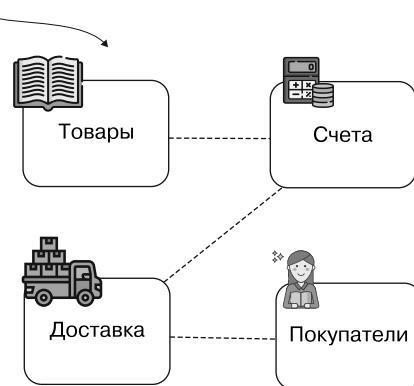


Рис. А.5. В SOA каждая функция представляет собой независимый процесс. Благодаря этому можно масштабировать только те функции, которые требуют больше ресурсов

Монолитная архитектура



Сервис-ориентированная архитектура



Рис. А.6. Монолитная система состоит из единственного приложения, и если систему разрабатывает несколько групп программистов, все они работают над одним и тем же продуктом. Такой подход требует более сложной координации. В случае SOA система состоит из нескольких приложений, так что каждая группа работает над своими частями. Это требует меньше координации между разработчиками

На первый взгляд все просто. Почему бы, учитывая преимущества, не создавать все приложения такими с самого начала? Зачем вообще утруждаться и говорить, что монолитная архитектура в некоторых случаях является подходящим решением? Чтобы ответить на эти вопросы, рассмотрим сложности, связанные с использованием SOA. Вот некоторые из областей, где при использовании SOA возникают проблемы.

1. Взаимодействие между сервисами.
2. Безопасность.
3. Хранение данных.
4. Развертывание системы.

Рассмотрим несколько примеров.

A.2.1. Усложнение взаимодействия между сервисами

Для реализации бизнес-потока по-прежнему необходимо взаимодействие между функциями. До сих пор, в монолитной архитектуре, данные функции были частью одного приложения, чтобы установить связь между ними, достаточно было вызвать соответствующий метод. Но, так как теперь это разные процессы, все усложнилось.

Сейчас функции должны взаимодействовать по сети. Обязательно помните следующее: сеть не является абсолютно надежной. Многие попадают в эту ловушку, забывая подумать, что случится, если в какой-то момент коммуникация между двумя компонентами нарушится. К сожалению, в отличие от монолитной архитектуры, в SOA любой вызов одного компонента другим может оказаться неудачным. В разных приложениях разработчики используют различные технологии и шаблоны для решения этой проблемы: например повторные вызовы, размыкатели цепочки и кэширование.

Второй момент, который необходимо учитывать, – есть различные способы установления связи между сервисами (рис. А.7): REST-сервисы, GraphQL, SOAP, gRPC, брокеры сообщений JMS, Kafka и др. Какой из них лучше? Разумеется, в любой ситуации подойдет один или несколько из предложенных вариантов. Во многих книгах вам встретятся пространные рассуждения и дискуссии по поводу того, как выбрать подходящий способ для типичных сценариев.

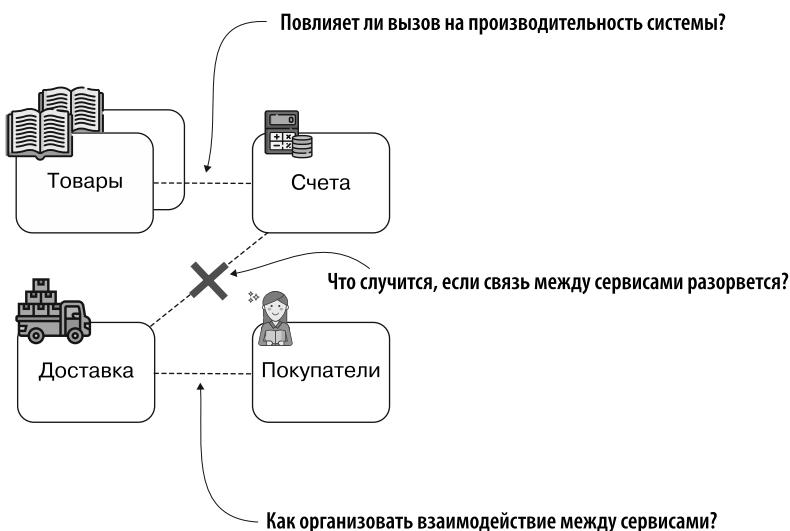


Рис. А.7. Взаимодействие между сервисами усложняет систему. Необходимо решить, как построить коммуникацию между ними. Следует также предусмотреть, что может случиться, если связь разорвется, и как решать возможные проблемы, вызванные нарушением контакта

A.2.2. Усложнение обеспечения безопасности системы

Распределив функционал между отдельными сервисами, мы также усложняем конфигурацию для обеспечения безопасности. Сервисы обмениваются сообщениями по сети, что чревато раскрытием информации. Некоторые данные (пароли, реквизиты банковских карт и т. д.) не должны стать общедоступными. Теперь, прежде чем их передавать, необходимо их зашифровать. Но даже если вам неважно, прочитает ли кто-нибудь эту информацию, все равно обычно нежелательно, чтобы кто-либо смог изменить данные, пока они передаются от одного компонента другому (рис. А.8).

A.2.3. Усложнение хранения данных

Большинству приложений необходимо как-то хранить информацию. Популярным способом для этого являются базы данных. В монолитных приложениях сведения хранятся в одной базе, как показано на рис. А.9. Подобную архитектуру называют *трехуровневой*, поскольку она состоит из трех слоев: клиентской части, серверной части и базы данных, используемой для хранения информации.



Рис. А.8. В SOA функции представляют собой отдельные сервисы, взаимодействующие по сети. Из-за этого появляется множество точек уязвимости. Разработчикам при построении приложения необходимо это учитывать

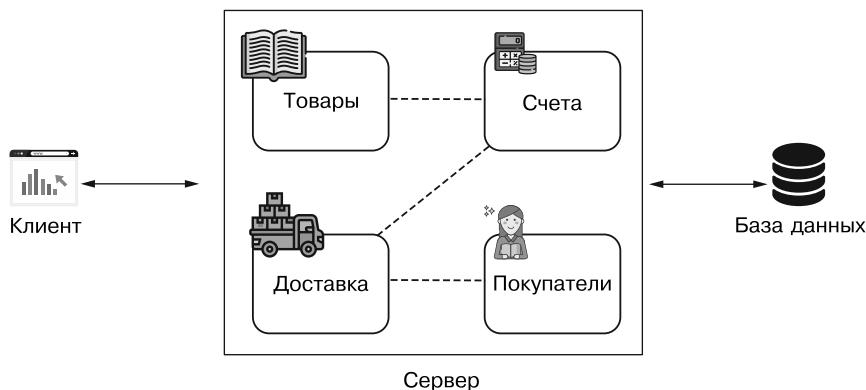


Рис. А.9. В монолитной архитектуре есть только одно приложение и, как правило, одна база данных. Это простая система, которую легко визуализировать и понять

В SOA есть несколько сервисов, нуждающихся в сохранении данных. И чем больше таких сервисов, тем больше вариантов структуры. Должна ли это быть одна база данных, общая для всех сервисов? Или стоит завести отдельную для каждого сервиса? Эти варианты показаны на рис. А.10.

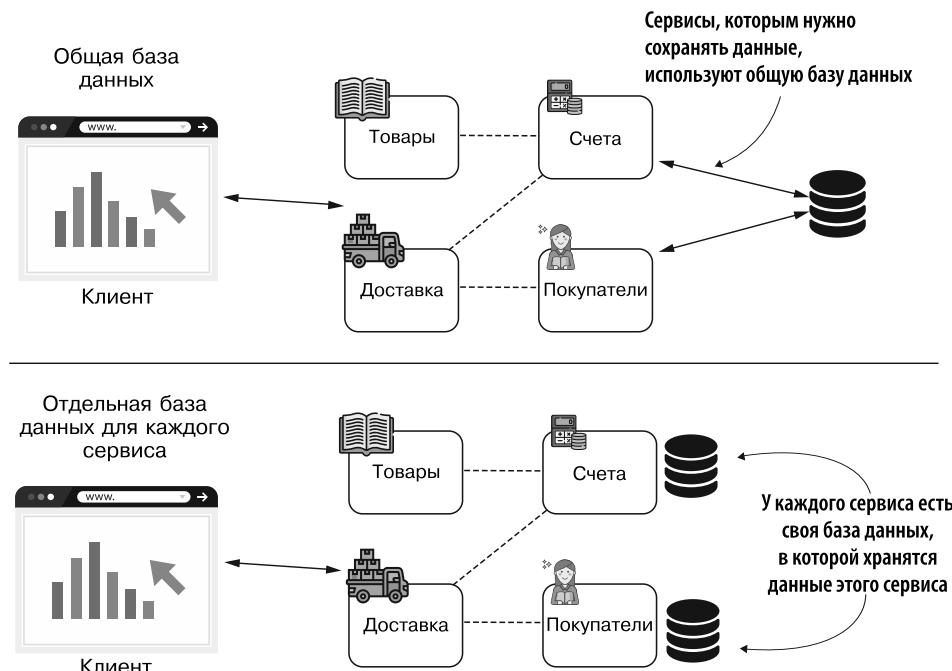


Рис. А.10. В SOA можно создать либо общую базу данных для всех сервисов, либо отдельную базу для каждого сервиса. У всех вариантов есть свои преимущества и недостатки — и это усложняет разработку уровня хранения данных в SOA

Большинство разработчиков считает, что использовать одну базу данных для всех сервисов — плохая идея. Исходя из моего личного опыта разделения монолитных приложений на сервисы, могу сказать, что общая база данных способна превратить разработку в сплошной кошмар. Но создание отдельной базы для каждого сервиса тоже сопряжено с трудностями. Как вы могли заметить при изучении транзакций, в случае одной базы гораздо проще гарантировать согласованность данных. При наличии нескольких независимых баз данных обеспечить согласованность между ними довольно сложно.

A.2.4. Усложнение развертывания системы

Пожалуй, проще всего предугадать, что SOA значительно усложняет развертывание системы. Теперь у нас есть не только множество сервисов, но и, как вы узнали из предыдущих абзацев, несколько баз данных. Если учесть еще и факт, что безопасность системы потребует более витиеватой конфигурации, станет ясно, насколько усложнится развертывание такой системы.

ЧТО ПЛОХОГО В МОНОЛИТЕ?

Как видите, SOA — это не всегда легко и просто. Возможно, теперь вы зааетесь вопросом: почему монолитную архитектуру считают чем-то плохим? На самом деле для многих систем она выглядит более осмысленной, чем SOA.

По моему мнению, причиной отрицательного отношения к монолитной архитектуре является то, что ее связывают со старыми системами. Большинство из них было создано прежде, чем кто-либо стал заботиться о принципах понятного кодирования и структурирования программ. Сейчас мы следим за тем, чтобы написанный код было проще поддерживать.

Сегодня это может показаться странным, но были времена, когда ничего подобного не существовало. Иногда мне даже встречаются разработчики, обвиняющие тех, кто стоял у истоков старых систем, в возникающих сегодня проблемах. На самом деле программисты не виноваты — они использовали инструменты и методики, которые на тот момент считались лучшими.

Сегодня многие разработчики утверждают, что сложный и плохо написанный код неразрывно связан с монолитной архитектурой. Но монолитные приложения могут быть модульными, а их код — чистым. И наоборот, сервис-ориентированные приложения могут быть запутанными и плохо структурированными.

A.3. ОТ МИКРОСЕРВИСОВ ДО БЕССЕРВЕРНЫХ ПРИЛОЖЕНИЙ

В своей книге я регулярно упоминаю микросервисы и хотел бы, чтобы вы имели хотя бы общее представление о значении данного слова. Микросервисы — это частный случай SOA. Обычно они разрабатываются таким образом, чтобы каждый микросервис соответствовал какой-то одной обязанности и имел собственное хранилище (микросервисы не используют общую базу данных).

Со временем способ развертывания приложений изменился. Архитектура приложений касается не только функциональности. Дальновидный разработчик программного обеспечения знает, как адаптировать архитектуру системы и к стилю работы группы программистов, и к способу развертывания системы. Возможно, вам приходилось слышать о движении DevOps, под которым подразумевается как способ развертывания программного обеспечения, так и способ его разработки.

В настоящее время приложения развертываются в облаке, с использованием виртуальных машин или контейнерных окружений — эти технологии обычно требуют, чтобы приложения были как можно меньшего размера. И тут эволюция, конечно же, подбросила новую неопределенность: насколько маленьким должен быть сервис? Этот вопрос горячо обсуждался в многочисленных книгах, статьях и дискуссиях.

Минимизация сервисов дошла до того, что сегодня можно реализовать короткую функцию, состоящую из нескольких строк кода, и развернуть ее в среде. Эта функция может активироваться и выполняться после таких событий, как HTTP-запрос, срабатывание таймера или передача сообщения. Подобные мелкие реализации называют *бессерверными* функциями. Слово «бессерверный» не означает, что функция выполняется вне сервера. Но, поскольку мы не видим ничего касающегося разработки (перед нами лишь код, реализующий логику функции, и события, которые ее активируют), все выглядит так, будто сервера не существует.

A.4. ЧТО ЕЩЕ ПОЧИТАТЬ

Архитектура программного обеспечения и ее эволюция — предмет в равной степени сложный и фантастический. Не думаю, что для наиболее полного раскрытия темы когда-либо будет написано слишком много книг. Я добавил этот раздел сюда, чтобы при встрече с теми или иными терминами вы понимали, о чем речь. Но на случай, если вы захотите глубже изучить вопрос, вот список изданий с моей книжной полки. Я перечисляю их в том порядке, в котором советую их читать.

1. *Bruce M., Pereira P. A. Microservices in Action* (Manning, 2018). Отличная книга, с которой можно начать изучение микросервисов. В ней вы найдете все основные темы, касающиеся микросервисов, с полезными примерами.
2. *Ричардсон К.* Микросервисы. Паттерны разработки и рефакторинга (Питер, 2019). Рекомендую перейти к данному изданию после того, как вы внимательно прочитаете *Microservices in Action*. Автор предлагает практический подход к разработке готовых к эксплуатации приложений с использованием микросервисов.
3. *Carnell J., Sánchez I. H. Spring Microservices in Action* (Manning, 2020). Эта книга поможет вам лучше понять, как использовать Spring для построения микросервисов.
4. *Siriwardena P., Dias N. Microservices Security in Action* (Manning, 2020). Здесь подробно рассказывается о том, что представляет собой безопасность в архитектуре микросервисов. Безопасность — критически важный аспект любой

системы, который необходимо принимать во внимание уже на самых первых этапах разработки. В книге данная тема раскрывается с нуля: прочитав ее, вы лучше поймете, какие моменты следует учитывать для обеспечения защиты микросервисов.

5. *Ньюмен С.* Создание микросервисов (Питер, 2016). В этом издании описаны шаблоны преобразования монолитной архитектуры в микросервисы. Обсуждается также, в каких случаях необходимо использовать микросервисы и на что нужно обратить внимание, принимая решение об их применении.

Приложение Б

Использование XML

в конфигурации контекста

Давным-давно, когда я только начинал использовать Spring, разработчики описывали конфигурацию контекста и фреймворка Spring в целом с помощью XML. Сегодня конфигурации, написанные на XML, встречаются разве что в старых, но до сих пор поддерживаемых приложениях. Разработчики перестали использовать язык XML в таких случаях уже много лет назад, заменив его аннотациями, из-за сложностей в чтении кода. Несмотря на то что у XML действительно есть ряд преимуществ, аннотации гораздо более пригодны для упрощения работы с кодом и поддержки приложения. Поэтому я решил не описывать XML-конфигурации в этой книге.

Если вы только начинаете работать со Spring, мой вам совет: изучайте XML-конфигурации, только если знаете, что вам придется поддерживать старый проект и других вариантов работы с ним вам не найти. Для начала ознакомьтесь с принципами, представленными в данной книге. Вы сможете использовать эти навыки для любой конфигурации, включая XML. Единственное различие будет в синтаксисе. Но если XML-конфигурации никогда не встретятся в вашей работе, рассматривать их нет смысла.

Для лучшего понимания того, что значит использовать XML-конфигурацию, я покажу вам, как добавить бин в контекст Spring этим устаревшим способом. Данный пример находится в проекте sq-app2-ex1.

Первое отличие, касающееся XML, состоит в том, что для размещения конфигурации вам потребуется отдельный файл (на самом деле это может быть несколько файлов, но я не буду углубляться в излишние подробности). Я назвал этот файл `config.xml` и поместил его в папке ресурсов проекта Maven. Вот содержимое файла:

```

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="parrot1" class="main.Parrot"> ← Создаем бин типа Parrot
        <property name="name" value="Kiki" /> ← с идентификатором parrot1
    </bean>
</beans>                                | Присваиваем этому
                                            | попугаю имя Kiki

```

Корневым тегом в этом файле является тег `<beans>`. Внутри него находится описание бина типа `Parrot`. Как показано в предыдущем фрагменте кода, чтобы определить бин, нужно использовать еще один XML-тег — `<bean>`. А чтобы присвоить имя попугая данному экземпляру, используется XML-тег `<property>`. В этом и состоит принцип XML-конфигурации: применение различных XML-тегов для определения тех или иных свойств. Там, где мы бы поставили аннотации, здесь будут XML-теги.

Мы создадим в классе `main` экземпляр, представляющий собой контекст Spring, и убедимся, что фреймворк добавил бин в контекст, обратившись к экземпляру типа `Parrot` и выведя в консоль имя попугая. В следующем фрагменте кода показана реализация метода `main`. Для создания экземпляра контекста Spring мне понадобился еще один класс.

При создании экземпляра контекста Spring с помощью класса `ClassPathXmlApplicationContext` нужно также сообщить, где находится файл `config.xml`, в котором содержится XML-конфигурация:

```

public class Main {

    public static void main(String[] args) {
        var context = new ClassPathXmlApplicationContext("config.xml");
        Parrot p = context.getBean(Parrot.class);

        System.out.println(p.getName());
    }
}

```

Когда вы запустите приложение, в консоли появится имя, присвоенное экземпляру попугая в XML-конфигурации (в моем случае это `Kiki`).

Приложение B

Краткое введение в HTTP

Далее мы поговорим об основных концепциях HTTP, которые должен знать каждый разработчик. К счастью, чтобы создавать веб-приложения, не обязательно быть экспертом по HTTP и помнить наизусть всю его документацию. Продвигаясь дальше по пути разработки программного обеспечения, вы со временем познакомитесь и с другими аспектами HTTP, но сейчас я бы хотел предоставить вам информацию, необходимую для понимания рассматриваемых, начиная с главы 7, примеров этой книги.

Зачем изучать HTTP, если издание посвящено Spring? Дело в том, что большинство современных приложений, которые вы будете создавать на основе фреймворков, — это веб-приложения, а там используется HTTP.

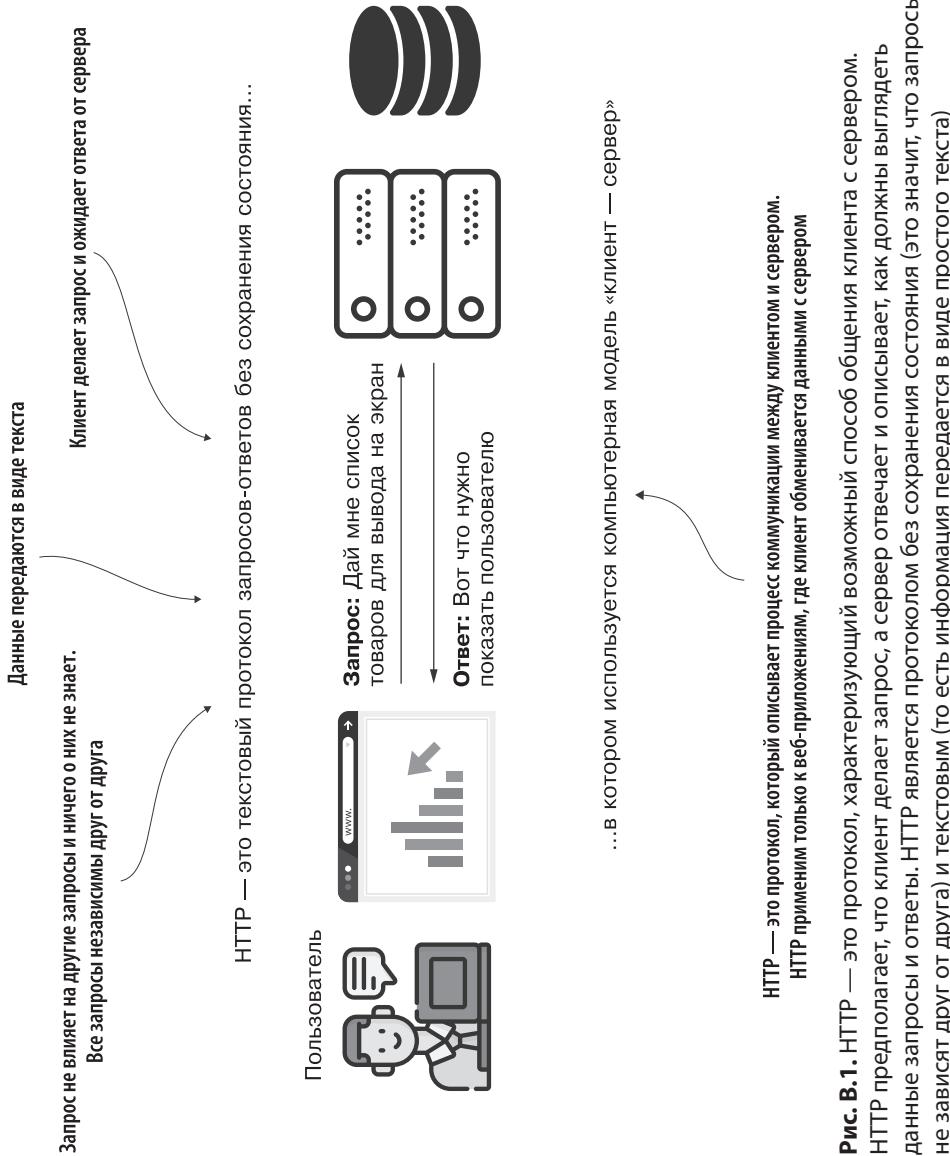
Мы начнем с выяснения, что такое HTTP, и проанализируем его определение с помощью диаграммы. Затем рассмотрим некоторые моменты, необходимые для понимания HTTP-запросов, создаваемых клиентом, и механизма, по которому сервер отвечает на эти запросы.

B.1. ЧТО ТАКОЕ HTTP

Я предпочитаю простые определения, поэтому скажу так: HTTP — это протокол, описывающий то, как клиент веб-приложения обменивается данными с сервером. В приложениях предпочитают использовать жесткие способы «говорить» с сервером, а протоколы предлагают правила обмена информацией. Проанализируем определение HTTP с помощью диаграммы на рис. B.1.

HTTP

Текстовый протокол запросов-ответов без сохранения состояния, в котором используется компьютерная модель «клиент — сервер».



B.2. HTTP-ЗАПРОСЫ КАК ЯЗЫК ОБЩЕНИЯ МЕЖДУ КЛИЕНТОМ И СЕРВЕРОМ

В приложениях, создаваемых на основе Spring, вам придется использовать HTTP-запросы для передачи данных от клиента серверу. При разработке клиентской части вы будете размещать данные в HTTP-запросе. При работе с сервером вам понадобится извлекать данные из запросов. В любом случае необходимо понимать, что такое HTTP-запрос.

Формат HTTP-запроса очень простой. Вам понадобится учитывать следующие его части.

1. *URI запроса* — путь, с помощью которого клиент сообщает серверу, какой ресурс он запрашивает. URI запроса выглядит примерно так: `http://www.manning.com/books/spring-start-here`.
2. *Метод запроса* — ключевое слово, используя которое клиент показывает, какое действие нужно выполнить с запрашиваемым ресурсом. Например, если написать адрес в адресной строке веб-браузера, браузер всегда применит HTTP-метод GET. В других случаях, как вы узнаете в следующих абзацах, клиент может использовать HTTP-запрос с такими методами, как POST, PUT или DELETE.
3. *Параметры запроса* — небольшое количество данных, которые клиент передает серверу в запросе. Говоря о небольшом количестве, я имею в виду что-то, что укладывается в 10–50 символов. Параметры (аргументы) запроса не являются обязательными. Они передаются в конце URI в виде выражения запроса.
4. *Заголовки запроса* — небольшое количество данных, передаваемых в заголовке запроса. В отличие от параметров запроса, эти значения не отображаются в URI. Заголовки также являются необязательным элементом.
5. *Тело запроса* — более значительное количество данных (несколько сотен символов), которые клиент передает серверу в рамках запроса. Не является обязательным элементом запроса.

Составные части HTTP показаны в следующем примере:

```
POST /servlet/default.jsp HTTP/1.1 ← Запрос характеризуют метод и путь
```

```
Accept: text/plain; text/html
Accept-Language: en-gb
Connection: Keep-Alive
Host: localhost
Referer: http://localhost/ch8/SendDetails.html
User-Agent: Mozilla/4.0 (MSIE 4.01;Windows 98)
Content-Length: 33
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate
```

В качестве данных
запроса могут быть
добавлены различные
заголовки и их значения

```
lastName=Einstein&firstName=Albert ← Для передачи данных запроса можно  
также использовать параметры запроса
```

URI запроса определяет ресурс на стороне сервера, с которым хочет работать клиент. URI — это та часть HTTP-запроса, о которой знают все — именно ее мы вводим в адресной строке браузера всякий раз, когда хотим открыть веб-сайт. Формат URI показан в следующем примере. Здесь <местоположение_сервера> — это сетевой адрес компьютера, на котором работает серверная часть приложения, <порт_приложения> — номер порта, через который доступен экземпляр серверного приложения, а <путь_ресурса> — путь, который разработчик связал с определенным ресурсом. Для работы с конкретным ресурсом клиент всегда должен запрашивать его путь:

```
http://<местоположение_сервера>:<порт_приложения>/<путь_ресурса>
```

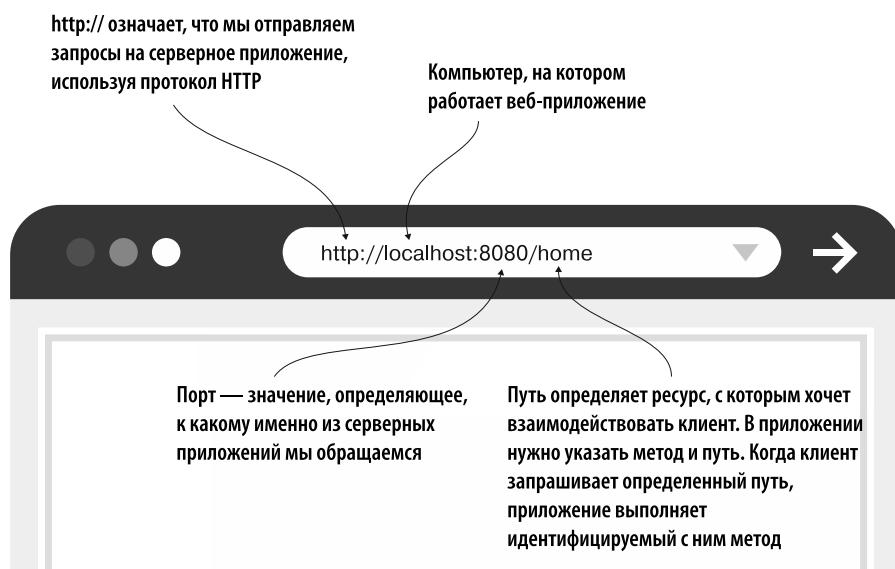


Рис. В.2. URI HTTP-запроса идентифицирует ресурс, который запрашивает клиент для дальнейшего взаимодействия с ним. Первая часть URI определяет протокол и сервер, на котором работает серверная часть приложения. Путь определяет ресурс, предоставляемый сервером

На рис. В.2 проанализирован формат URI для HTTP-запроса.

ПРИМЕЧАНИЕ

Универсальный идентификатор ресурса (Uniform Resource Identifier, URI) состоит из универсального локатора ресурса (Uniform Resource Locator, URL) и пути. Можно сказать, что URI вычисляется по формуле $URI = URL + \text{путь}$. Но люди часто путают URI и URL или считают, что это одно и то же. Нужно помнить, что URL идентифицирует сервер и приложение. Если к URL добавить путь к определенному ресурсу приложения, получим URI.

После того как клиент идентифицирует ресурс и запрос, он определяет, что будет делать с данным ресурсом. Для этого клиент использует ключевое слово, называемое *методом HTTP-запроса*. Способ, которым клиент определяет метод, зависит от того, как он обращается к серверу. Например, если вызов сервера поступает непосредственно из браузера, когда пользователь вводит адрес в адресную строку, то браузер отправляет GET-запрос. В большинстве случаев, когда вы нажимаете на кнопку отправки формы на веб-странице, браузер использует метод POST. Метод, который в такой ситуации должен применяться в браузере, выбирает разработчик веб-страницы. Подробнее вы узнаете об этом в главе 8. HTTP-запрос также может быть отправлен сценарием, написанным на клиентском языке, например JavaScript. В этом случае решение об HTTP-методе, используемом при запросе, принимает разработчик сценария.

В веб-приложениях используются главным образом следующие HTTP-методы:

- *GET* — отражает намерение клиента получить с сервера какие-то данные;
- *POST* — отражает намерение клиента загрузить данные на сервер;
- *PUT* — отражает намерение клиента изменить данные, хранящиеся на сервере;
- *DELETE* — отражает намерение клиента удалить данные с сервера.

ПРИМЕЧАНИЕ

Всегда следует помнить, что ключевое слово не накладывает ограничений на то, что вы делаете. Протокол HTTP не может заставить вас не изменять данные на стороне сервера при использовании HTTP-метода GET. Но никогда не применяйте HTTP-методы не по назначению! Всегда учитывайте особенности конкретного HTTP-метода, чтобы сохранить надежность, безопасность и удобство обслуживания приложения.

Следующие HTTP-методы не столь популярны, но тоже встречаются достаточно часто:

- *OPTIONS* — сообщает серверу, что нужно вернуть список параметров, поддерживаемых для обработки запросов. Например, клиент может захотеть узнать, какие HTTP-методы поддерживает сервер. Как правило, метод OPTIONS используется в функциях обмена ресурсами с запросом происхождения (Cross-Origin Resource Sharing, CORS), связанных со средствами обеспечения безопасности. Отличное обсуждение CORS вы найдете в главе 10 еще одной моей книги, *Spring Security in Action* (Manning, 2020; <https://livebook.manning.com/book/spring-security-in-action/chapter-10/>);
- *PATCH* — может использоваться при необходимости изменить лишь часть данных, предоставляемых определенным ресурсом на сервере. HTTP-метод PUT применяется только в тех случаях, когда действие клиента полностью заменяет выбранный ресурс или даже создает его, если данные, которые нужно модифицировать, еще не существуют. Как показывает мой опыт, разработчики стараются использовать по большей части HTTP-метод PUT, даже если на самом деле действие соответствует методу PATCH.

URI и HTTP-метод являются обязательной частью запроса. Формируя HTTP-запрос, клиент должен указать ресурс, с которым он собирается взаимодействовать (URI), и желаемое действие с этим ресурсом (HTTP-метод).

Например, запрос, представленный в следующем фрагменте, может сообщать серверу, что нужно вернуть все товары (*products*), контролируемые приложением. Предполагается, что *products* является ресурсом, которым управляет сервер:

```
GET http://example.com/products
```

Следующий запрос может означать, что клиент хочет удалить все товары с сервера:

```
DELETE http://example.com/products
```

Но иногда клиенту также нужно передать в запросе какие-либо данные, необходимые серверу для выполнения этого запроса. Предположим, клиент хочет удалить не все товары, а только один из них. Тогда он должен сообщить серверу, какой именно товар следует убрать, и передать эти данные в запросе. Подобный HTTP-запрос может выглядеть так, как показано ниже. Здесь клиент с помощью параметра объявляет, что нужно удалить товар *Beer*:

```
DELETE http://example.com/products?product=Beer
```

Клиент также может использовать *параметры запроса*, *заголовки запроса* или *тело запроса*. Параметры и тело запроса не являются обязательными частями HTTP-запроса. Клиент добавляет их только для того, чтобы передать на сервер определенные данные.

Параметры запроса представляют собой пары типа «ключ — значение». Клиент добавляет их в HTTP-запрос, чтобы отправить на сервер определенную информацию. Параметры используются для передачи небольших отдельных элементов данных. Если данных больше, лучше их поместить в теле HTTP-запроса. В главах 7–10 мы используем оба способа для передачи данных от клиента серверу в HTTP-запросе.

В.3. HTTP-ОТВЕТ: СПОСОБ ПОЛУЧИТЬ ОТВЕТ ОТ СЕРВЕРА

HTTP — это протокол, который позволяет клиенту взаимодействовать с сервером в рамках веб-приложения. Запросы клиента приложения мы уже рассмотрели, теперь пора заняться ответами сервера. На запрос клиента сервер может отправить следующее.

- *Статус ответа* — целое число от 100 до 599, кратко представляющее результат запроса.
- *Заголовки ответа (необязательные)* — данные в виде пар «ключ — значение», подобно параметрам запроса. Предназначены для передачи небольших

объемов информации (от 10 до 50 знаков) от сервера клиенту в ответ на запрос клиента.

- *Тело ответа (необязательное)* — способ передать много данных с сервера в ответ на запрос клиента (например, если сервер должен отправить несколько сотен символов или даже несколько файлов).

Следующий пример кода поможет вам лучше понять, что собой представляет HTTP-ответ:

HTTP/1.1 200 OK ←———— В HTTP-ответе указывается версия HTTP, код ответа и сообщение

```
Server: Microsoft-IIS/4.0
Date: Mon, 14 May 2012 13:13:33 GMT
Content-Type: text/html
Last-Modified: Mon, 14 May 2012 13:03:42 GMT
Content-Length: 112
```

HTTP-ответ может содержать заголовки, где передаются данные

```
<html>
<head><title>HTTP Response</title></head>
<body>Hello Albert!</body>
</html>
```

HTTP-ответ может содержать тело, в котором отправляются данные

Статус ответа — единственная обязательная часть HTTP-ответа, передаваемого сервером на запрос клиента. Статус сообщает клиенту, понял ли сервер запрос, все ли сделано правильно или же в процессе обработки запроса возникли проблемы. Например, если сервер возвращает значение статуса, начинающееся с 2, это означает, что все хорошо. HTTP-статус — это краткое представление результата полного запроса (включая информацию о том, выполнил ли сервер бизнес-логику запроса). Запоминать подробности всех статусов не обязательно. Я перечислю и опишу только те из них, которые чаще всего будут встречаться вам на практике.

Статусы, начинающиеся с 2, показывают, что сервер понял запрос правильно. Запрос успешно обработан, сервер выполнил все, чего хотел клиент.

Если статус начинается с 4, сервер сообщает клиенту, что с запросом что-то не так и проблема лежит на стороне клиента. Например, клиент может запрашивать несуществующий ресурс или передавать параметры запроса, которые не ожидаются сервером.

Статус начинается с 5, если сервер сообщает о проблеме на своей стороне. Например, серверу нужно было установить соединение с базой данных, но она оказалась недоступна. В этом случае сервер возвращает статус, в котором объявляет клиенту, что не может выполнить запрос, но не потому, что клиент что-то сделал не так, а по своим причинам.

ПРИМЕЧАНИЕ

Я пропускаю значения от 1 до 3, поскольку они не особенно часто встречаются в приложениях — так вы сможете сосредоточиться на трех наиболее важных категориях.

Значения, начинающиеся с 2, — это разные варианты сообщений о корректной обработке запроса клиента. Вот несколько примеров подобных статусов:

- 200 — **OK**. Самый известный и самый простой статус ответа. Он просто сообщает клиенту, что на сервере не возникло проблем при обработке запроса;
- 201 — **Created**. Может использоваться, например, в ответе на POST-запрос, как свидетельство, что серверу удалось создать требуемый ресурс. Раскрывать такие подробности в статусе ответа не всегда обязательно — именно поэтому значение, указанное в пункте выше, является наиболее частым способом показать, что все в порядке;
- 204 — **No Content**. Так можно объявить клиенту, что не следует ожидать данных в теле этого ответа сервера.

Если статус HTTP-ответа начинается с 4, сервер говорит клиенту, что с запросом что-то не так. Например, клиент мог что-то напутать, запрашивая определенный ресурс. Возможно, этот ресурс не существует (всем известный статус 404 — *Not Found*), или какие-то данные не прошли проверку. Вот некоторые из наиболее часто встречающихся значений, сообщающих об ошибках клиента:

- 400 — **Bad Request**. Обобщенный статус, применяемый для обозначения любого рода проблемы, связанной с HTTP-ответом (такой как проверка данных или невозможность прочесть определенное значение в теле или параметрах запроса);
- 401 — **Unauthorized**. Это значение используется для сообщения о необходимости аутентификации запроса;
- 403 — **Forbidden**. Данный статус передается сервером как знак, что клиент не был авторизован для выполнения запроса;
- 404 — **Not Found**. Такое значение сообщает клиенту, что запрошенный ресурс не существует.

Если статус ответа начинается с 5, он показывает, что что-то пошло не так на стороне сервера — и это именно проблема сервера, а не клиента. Клиент передал корректный запрос, но сервер по какой-то причине не может его выполнить. Из статусов данной категории чаще всего встречается 500 — *Internal Server Error*. Это общее уведомление об ошибке. Оно отправляется, чтобы сообщить клиенту, что при обработке запроса на стороне сервера возникла какая-то проблема.

Если вы хотите глубже изучить коды статусов, будет полезно прочитать эту страницу: <https://datatracker.ietf.org/doc/html/rfc7231>.

В ответ на запрос сервер также может передавать клиенту данные — в заголовках или в теле ответа.

B.4. HTTP-СЕССИЯ

HTTP-сессия — механизм, позволяющий серверу хранить данные в течение нескольких взаимодействий типа «вопрос — ответ» с одним и тем же клиентом. Напомню, что по протоколу HTTP все запросы независимы друг от друга. Один запрос ничего не знает о другом запросе, поступающем до него, после него или вместе с ним. Запрос не может использовать данные другого запроса или получать доступ к информации, передаваемой с сервера в ответ на него.

Однако часто вам будут встречаться задачи, когда серверу необходимо согласовать некоторые запросы. Хороший пример — функционал корзины в онлайн-магазине. Пользователь помещает в корзину разные товары. Чтобы добавить товар в корзину, клиент делает запрос. Чтобы положить туда следующий продукт, клиент делает еще один запрос. Серверу нужен способ узнать, что этот клиент раньше уже добавлял товар в корзину (рис. B.3).

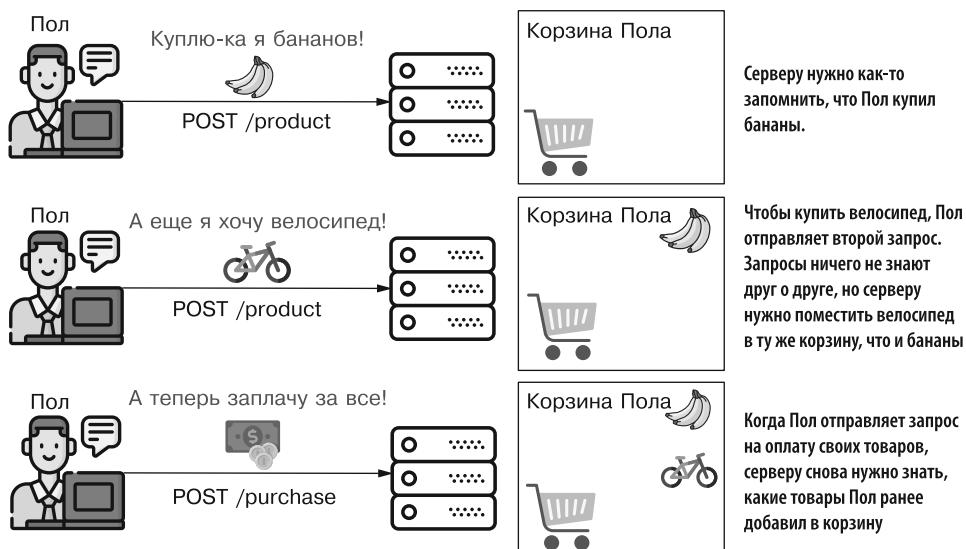


Рис. B.3. Серверная часть онлайн-магазина должна различать клиентов и помнить, какие товары они поместили в свои корзины. Поскольку HTTP-запросы не зависят друг от друга, серверу нужен другой способ зафиксировать товары, которые добавил в корзину каждый клиент

Один из способов реализовать подобное поведение — использовать HTTP-сессию. Бэкенд присваивает клиенту уникальный идентификатор, называемый «ID сессии», и связывает его с неким местом в памяти приложения. Каждый запрос, отправляемый клиентом после этого, должен содержать в заголовке ID сессии. Так, серверная часть приложения может связать все запросы, которые производятся в рамках данной сессии (рис. B.4).

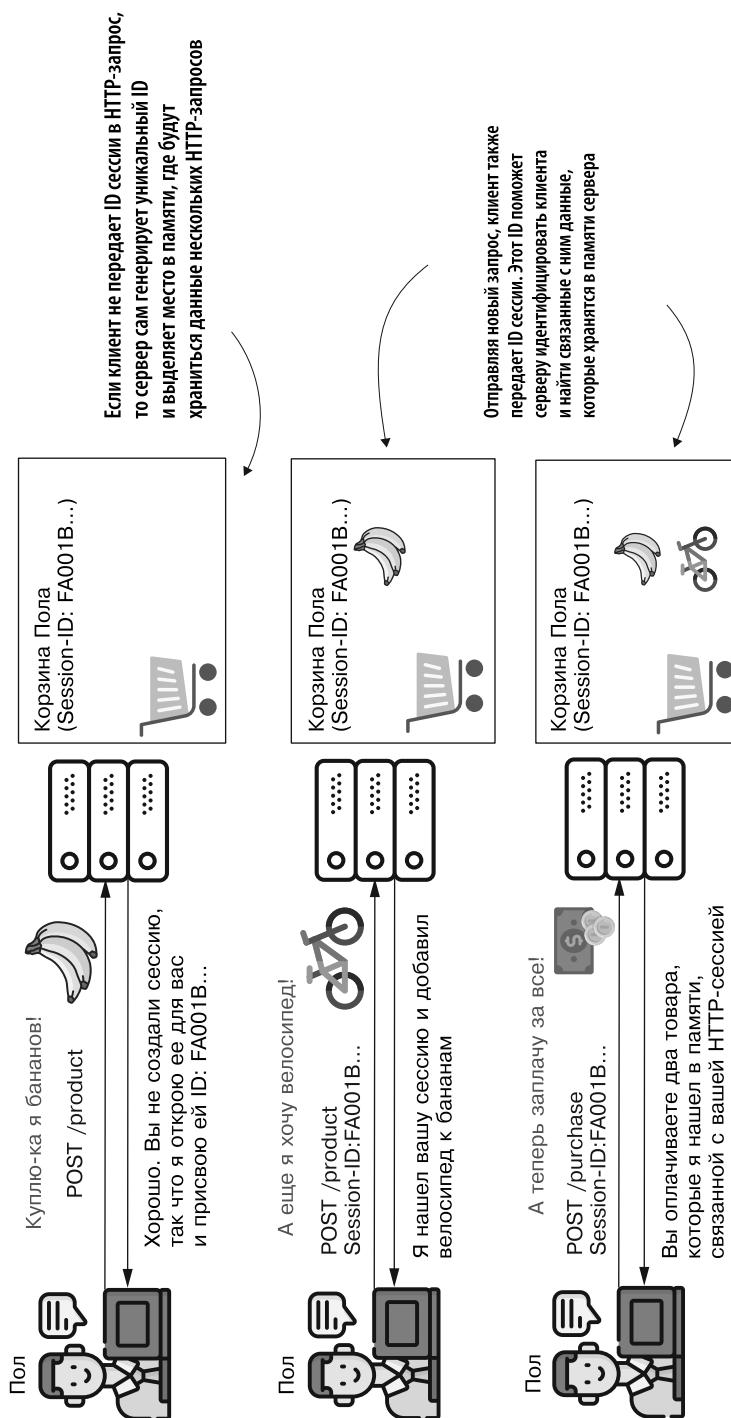


Рис. B.4. Механизм работы HTTP-сессии. Клиент идентифицируется по уникальному ID сессии, сгенерированному сервером. В нескольких последующих запросах клиент передает этот ID, благодаря чему серверная часть приложения знает, какая именно память была выделена для него ранее

Обычно HTTP-сессия закрывается через какое-то время после того, как клиент перестает отправлять запросы. Вы можете обозначить данное время (как правило, это делается в контейнере сервлетов и в приложении). Сессия не должна длиться больше чем несколько часов. Если она продолжается слишком долго, сервер тратит на нее много памяти. В большинстве приложений, если клиент в это время перестает отправлять запросы, сессии завершаются меньше чем за час.

Если после окончания сессии клиент отправляет следующий запрос, сервер начинает для него новую сессию.

Приложение Г

Представление данных в формате JSON

JSON (JavaScript Object Notation) — распространенный способ представления данных, используемый для их передачи между приложениями в запросах и ответах по протоколу HTTP при коммуникации через конечные точки REST (рис. Г.1). Поскольку конечные точки REST — один из самых распространенных способов установить взаимодействие между приложениями, а JSON — основной формат отправляемых данных, очень важно знать этот формат и уметь им пользоваться.

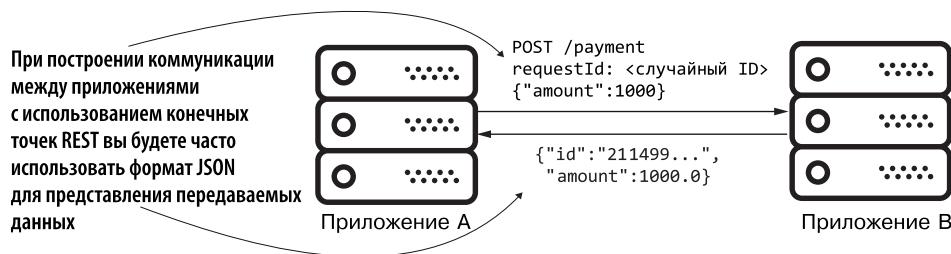


Рис. Г.1. Разработка бизнес-логики часто подразумевает построение коммуникации между несколькими приложениями. Обмен данными между ними обычно происходит в формате JSON. Вам необходимо знать JSON для реализации и тестирования конечных точек REST

К счастью, JSON — очень простой формат, в нем всего несколько правил. Главное, что вам нужно понять, — как в JSON представляются экземпляры объектов с использованием их атрибутов. Подобно остальным элементам класса Java, атрибуты описываются именами и текущими значениями. Можно сказать, что

у объекта `Product` есть атрибут `name` и атрибут `price`. Значением `name` может быть, например, `chocolate`, а значением `price` — 5. Чтобы представить это в формате JSON, нужно учесть следующее:

- экземпляры объектов в JSON помещаются в фигурные скобки;
- внутри фигурных скобок перечисляются пары «атрибут — значение», разделенные запятыми;
- имена атрибутов заключаются в двойные кавычки;
- строковые значения пишутся в двойных кавычках (если двойные кавычки содержатся в самой строке, то перед ними ставится обратная косая черта (\));
- числовые значения пишутся без кавычек;
- между именем и атрибутом ставится двоеточие.

На рис. Г.2 показан экземпляр объекта `Product` в формате JSON. Атрибут `name` этого объекта имеет значение `chocolate`, а атрибут `price` — 5.



Рис. Г.2. Представление экземпляра объекта в формате JSON. Пары «атрибут — значение» заключаются в фигурные скобки. Между атрибутом и его значением ставится двоеточие. Сами пары перечисляются через запятую

В JSON сам по себе объект не может иметь имени или типа. Ниоткуда не следует, что данный фрагмент описывает именно товар. Единственными значимыми элементами объекта JSON являются атрибуты. Элементы формата JSON и правила представления объекта показаны на рис. Г.2.

Внутри одного объекта может содержаться экземпляр другого объекта как значение одного из атрибутов. Если в классе `Product` есть атрибут типа `Pack`, где `Pack` — объект, описываемый атрибутом `color`, представление экземпляра `Product` может выглядеть так:

```
{
    "name" : "chocolate",
    "price" : 5,
    "pack" : { ← Значением атрибута pack является экземпляр объекта
        "color" : "blue"
    }
}
```

Здесь действуют все те же правила. Другой объект тоже может описываться несколькими атрибутами. Один объект может находиться внутри другого, и так сколько угодно раз.

Чтобы определить коллекцию объектов JSON, нужно заключить ее в квадратные скобки, разделив элементы запятыми. Ниже показана коллекция, состоящая из двух элементов типа `Product`:

```
[ ← Экземпляры объектов, образующие коллекцию, заключаются в фигурные скобки
{
    "name" : "chocolate",
    "price" : 5
},
← Экземпляры разделяются запятыми
{
    "name" : "candy",
    "price" : 3
}]
```

Приложение Д

Установка MySQL

и создание базы данных

Далее я покажу вам, как создать базу данных MySQL. В отдельных примерах, с которыми мы работаем в главах 12–14, используется внешняя система управления базами данных (СУБД). Для выполнения этих примеров еще до разработки проекта понадобится создать свою базу данных — с ней и будет взаимодействовать приложение.

Выбор технологий баз данных очень широк: MySQL, Postgres, Oracle, сервер MS SQL и т. п. Если у вас уже есть технология, которую вы предпочитаете использовать, советую взять именно ее. Для примеров, рассматриваемых в этой книге, мне тоже пришлось определиться с технологией, и я решил, что это будет MySQL — бесплатная, занимающая мало места и простая в установке для любой операционной системы. Именно MySQL применяется в большинстве учебных материалов и примеров.

При изучении Java и Spring не имеет значения, какую именно СУБД вы выберете. Независимо от технологии — MySQL, Oracle, Postgres или какой-либо еще — классы и методы Java и Spring будут одни и те же.

Для создания базы данных, которая будет использоваться в примерах, нужно выполнить следующие операции:

- 1) установите СУБД на свой локальный компьютер — нам понадобится MySQL;
- 2) установите клиентское приложение для СУБД — в нашем случае MySQL Workbench, одно из самых распространенных клиентских приложений для MySQL;
- 3) находясь в клиентском приложении, установите соединение с СУБД, размещенной на вашем компьютере;
- 4) создайте базу данных, с которой будете работать в примере.

ШАГ 1. УСТАНОВКА СУБД НА ЛОКАЛЬНОМ КОМПЬЮТЕРЕ

Прежде всего нужно убедиться, что у нас есть СУБД, с которой мы будем работать. В примерах этой книги используется MySQL, но если вам больше нравится другая технология, можете применить ее. Если выберете MySQL, скачать установочный пакет можно по этой ссылке: <https://dev.mysql.com/downloads/mysql/>.

Загрузите установочный пакет, соответствующий вашей операционной системе, и выполните операции по установке, которые описаны здесь: <https://dev.mysql.com/doc/refman/8.0/en/installing.html>.

Обратите внимание, что во время установки СУБД может потребоваться создать учетную запись. Запомните параметры доступа к ней — они вам понадобятся на шаге 3.

ШАГ 2. УСТАНОВКА КЛИЕНТСКОГО ПРИЛОЖЕНИЯ ДЛЯ ВЫБРАННОЙ СУБД

Для взаимодействия с СУБД вам понадобится клиентское приложение, с помощью которого вы будете создавать базу данных, время от времени изменять ее структуру и проверять работу приложения. Какое клиентское приложение установить — зависит от того, какую технологию СУБД вы выберете. Для MySQL можно использовать MySQL Workbench — один из самых распространенных клиентов MySQL.

Загрузите установочный пакет MySQL Workbench, соответствующий вашей операционной системе (<https://dev.mysql.com/downloads/workbench/>), и установите MySQL Workbench, как описано в руководстве: <https://dev.mysql.com/doc/workbench/en/wb-installing.html>.

ШАГ 3. УСТАНОВКА СОЕДИНЕНИЯ С ЛОКАЛЬНОЙ СУБД

После того как на локальном компьютере появится СУБД и клиент для нее, необходимо будет подключить этого клиента к СУБД. Для установки соединения вам понадобятся параметры доступа, выбранные на шаге 1. Если при установке СУБД у вас не спросили авторизационные данные, можете использовать имя пользователя `root` и пустой пароль.

Я покажу вам, как установить соединение для MySQL Workbench. После того как вы откроете MySQL Workbench, нужно щелкнуть на пиктограмме в виде знака плюс рядом с надписью MySQL Connections (рис. Д.1).



Рис. Д.1. Чтобы установить соединение с базой данных, откройте MySQL Workbench и щелкните на знаке плюс рядом с надписью MySQL Connections

Далее появится всплывающее окно, в котором нужно ввести имя соединения и параметры доступа, выбранные при установке СУБД. MySQL Workbench будет использовать эти параметры доступа для аутентификации в СУБД (рис. Д.2).

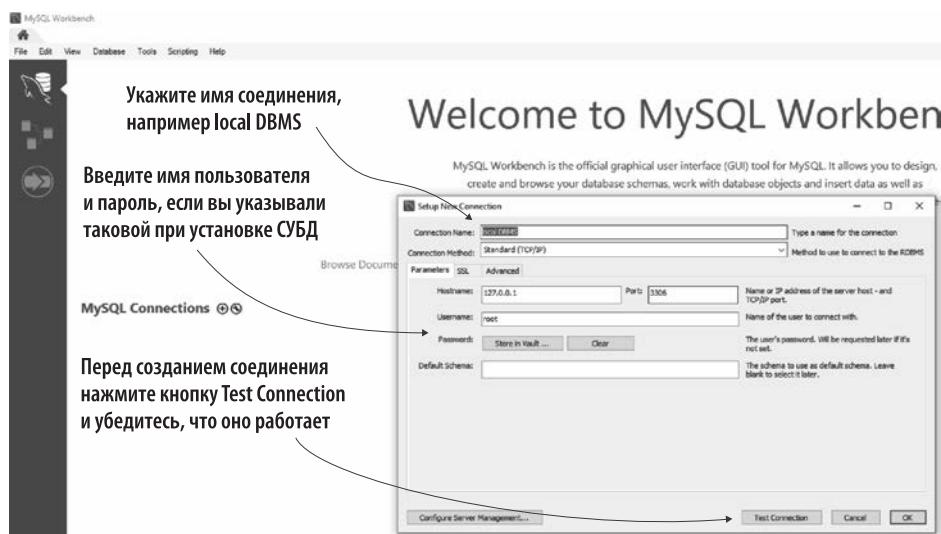


Рис. Д.2. Присвойте соединению имя и введите параметры доступа, выбранные при установке СУБД

Теперь нажмите кнопку Test Connection и убедитесь, что MySQL Workbench может установить соединение с базой данных.

Для некоторых версий MySQL в MySQL Workbench может появляться предупреждение. В работе с примерами книги это не имеет значения, так что, если увидите его, нажмите кнопку Continue Anyway (рис. Д.3).

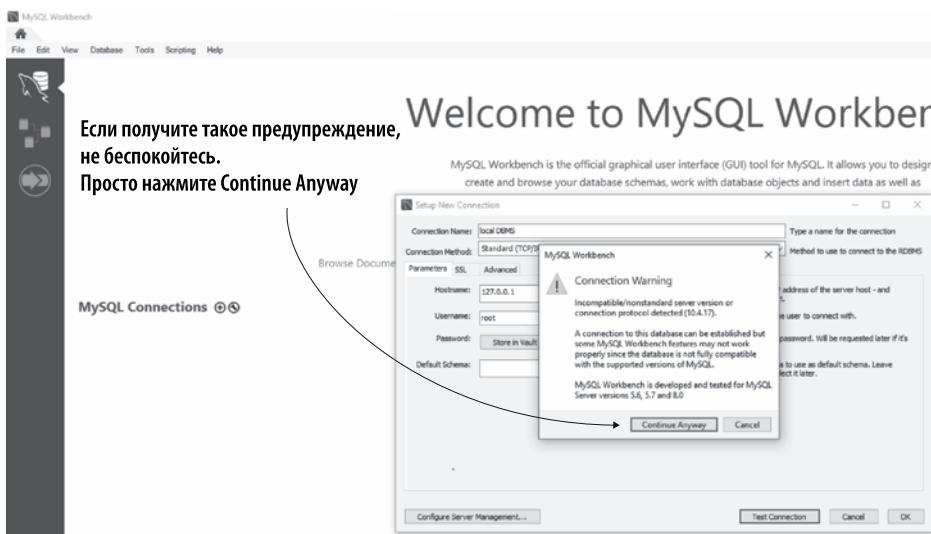


Рис. Д.3. Иногда MySQL Workbench выводит предупреждение. Для примеров из этой книги оно не имеет значения. Если увидите его, нажмите кнопку Continue Anyway

Если параметры соединения указаны правильно и клиент MySQL Workbench установил соединение с локальной СУБД, появится всплывающее окно, подобное тому, что показано на рис. Д.4.

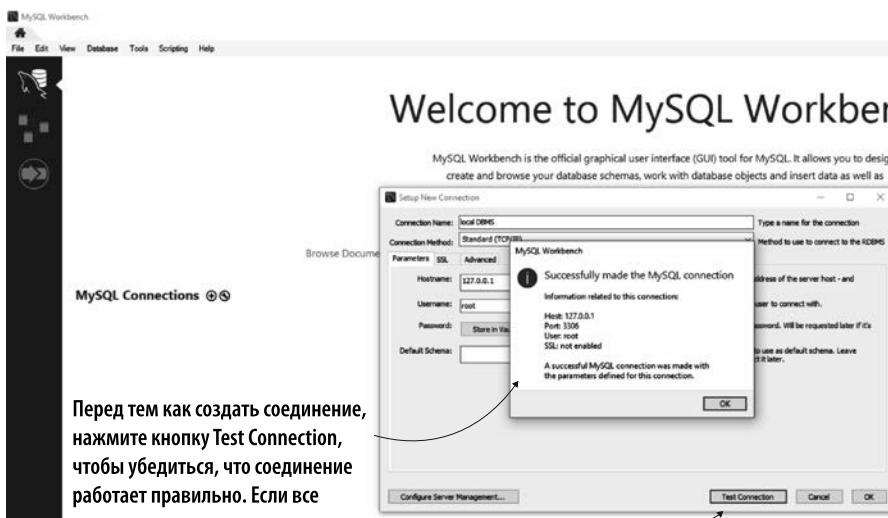


Рис. Д.4. При установке соединения MySQL Workbench с локальной СУБД появится всплывающее окно с сообщением Successfully made the MySQL connection

438 Приложение Д. Установка MySQL и создание базы данных

После того как вы нажмете кнопку OK, новое соединение будет создано. Вы увидите его в главном окне MySQL Workbench: оно будет изображено в виде прямоугольника с именем соединения внутри (рис. Д.5).



Рис. Д.5. Теперь в главном окне MySQL Workbench появится созданное вами соединение. Оно изображено в виде серого прямоугольника с именем соединения внутри

ШАГ 4. СОЗДАНИЕ БАЗЫ ДАННЫХ

После того как соединение будет установлено, можно создать с его помощью базу данных (она понадобится нам в примерах глав 12–14). Это нужно сделать еще до разработки приложения.

Щелкнув дважды на прямоугольнике, представляющем соединение с локальной СУБД в главном окне MySQL Workbench, вы увидите экран, подобный тому, что показан на рис. Д.6. Чтобы создать базу данных, нажмите на пиктограмму в виде маленького цилиндра, расположенную на панели инструментов.

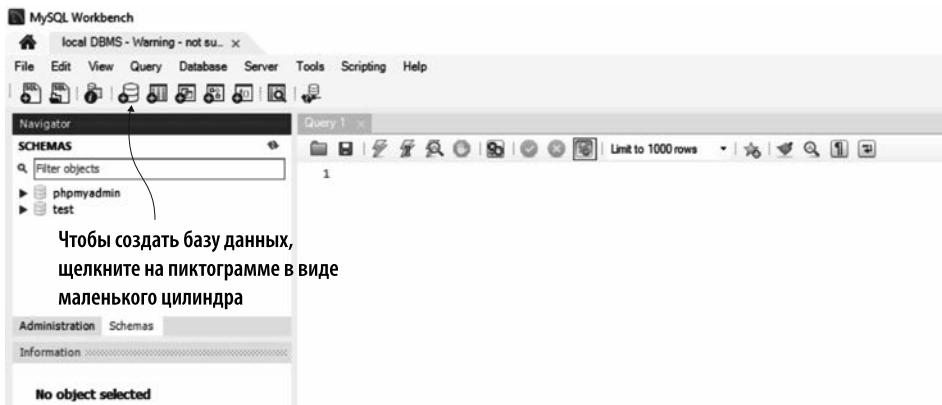


Рис. Д.6. После того как будет установлено соединение с СУБД, можно создать базу данных, щелкнув на пиктограмме в виде маленького цилиндра, расположенной на панели инструментов

Затем нужно присвоить базе имя и нажать кнопку **Apply**, как показано на рис. Д.7.

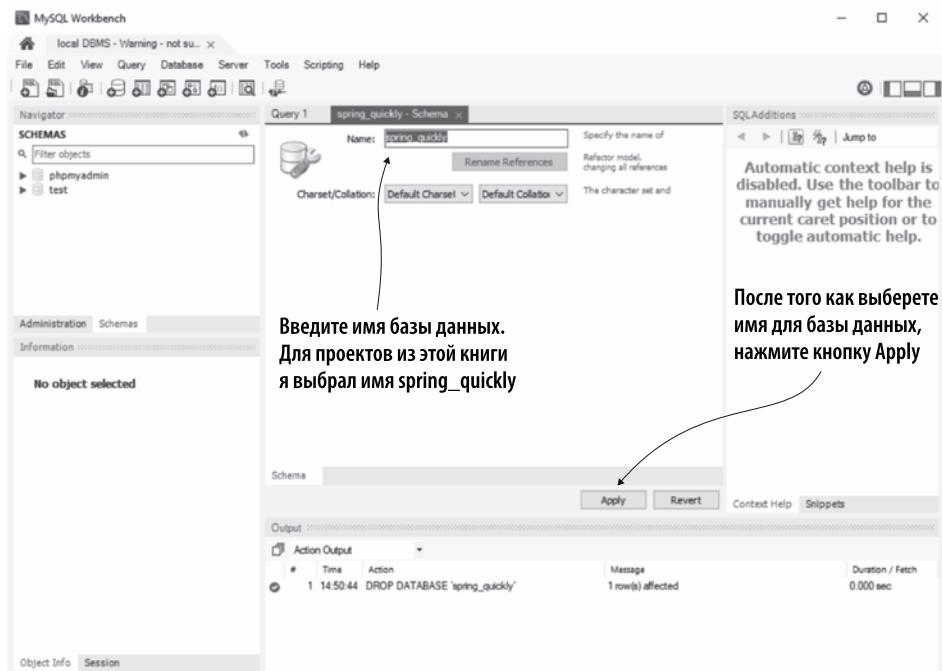
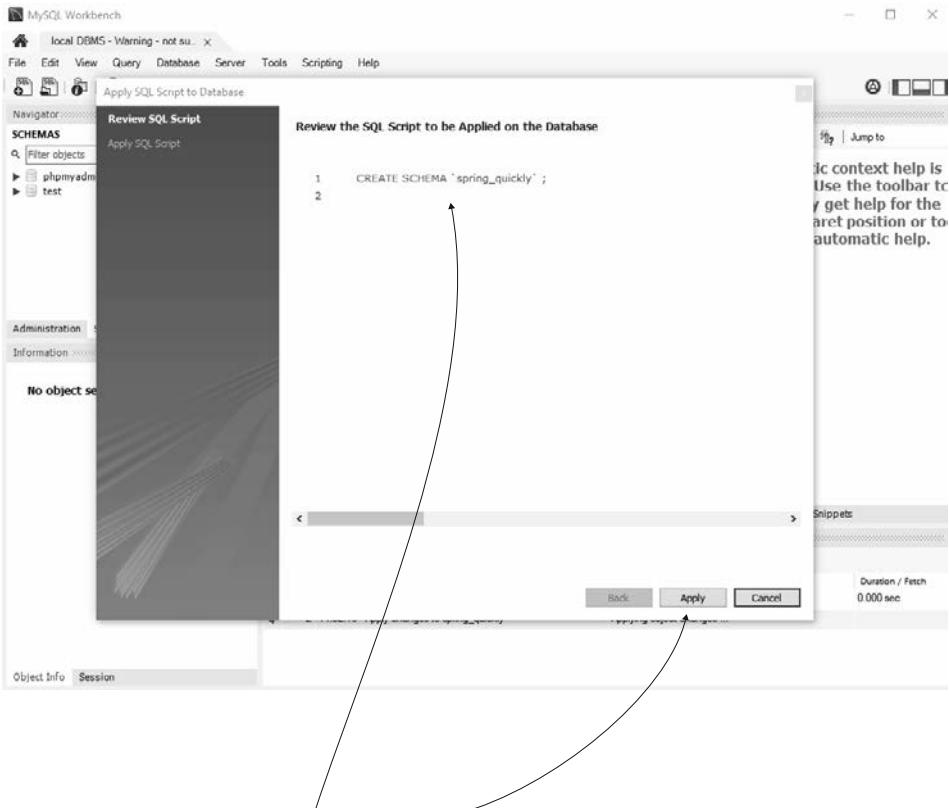


Рис. Д.7. Присвойте базе данных имя. В наших примерах это **spring_quickly**. Для создания базы данных нажмите кнопку **Apply**

Прежде чем создать базу данных, MySQL Workbench снова запросит подтверждения. Снова нажмите кнопку **Apply**, как показано на рис. Д.8.



В этом всплывающем окне выводится SQL-запрос, который MySQL Workbench отправит в СУБД для создания базы данных. Нажмите кнопку **Apply**, чтобы отправить этот SQL-запрос и создать базу данных

Рис. Д.8. Нужно снова подтвердить ваше намерение создать базу данных. В этом окне вы также увидите запрос, который MySQL Workbench отправит в СУБД для создания новой базы. Чтобы выполнить этот запрос и создать базу данных, нажмите кнопку **Apply**

Если имя новой базы данных появится на левой панели окна, это значит, что база создана успешно и ее можно использовать в приложениях Spring (рис. Д.9).

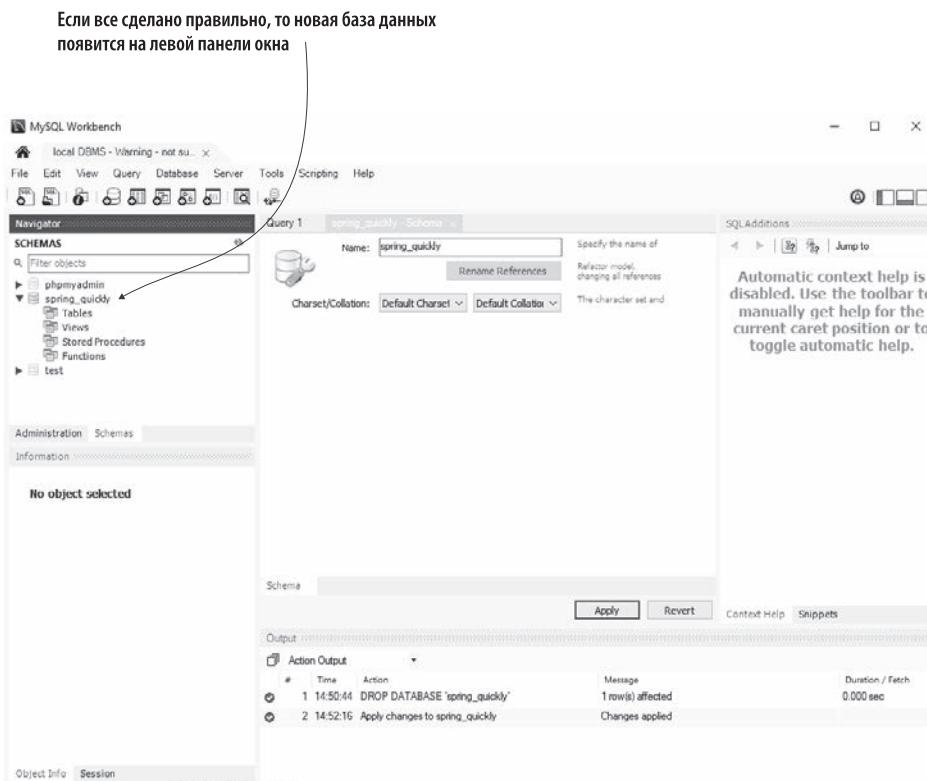


Рис. Д.9. Новая база данных появится на левой панели окна. Если вы ее там видите, значит, вы успешно создали базу данных

Приложение E

Рекомендованные инструменты

Далее я перечислю инструменты, использованные в примерах этой книги, а также их альтернативы, которые тоже могут оказаться полезными.

IDE

- *JetBrains IntelliJ IDEA*. Для примеров в своей книге я использовал IDE IntelliJ. Вы можете открывать эти примеры или собирать собственные в бесплатной редакции IntelliJ Community. Если удастся получить доступ к IntelliJ Ultimate, вы сможете попробовать многочисленные дополнительные элементы, упрощающие разработку Spring-приложений. Версия Ultimate является лицензированной. Подробнее об IntelliJ можно прочитать на официальной веб-странице IDE: <https://www.jetbrains.com/idea/>.
- *Eclipse IDE*. Eclipse – это IDE с открытым исходным кодом, которая является альтернативой для IntelliJ IDEA. Советую использовать Eclipse в сочетании со Spring Tools, чтобы получить более глубокий опыт разработки Spring-приложений. Узнать больше об Eclipse и загрузить данную IDE можно здесь: <https://www.eclipse.org/downloads/>.
- *Spring Tools*. Это набор инструментов, интегрируемых в IDE с открытым кодом (например Eclipse), чтобы упростить разработку Spring-приложений. Более детальную информацию о Spring Tools можно получить на официальной странице инструментария: <https://spring.io/tools>.

Инструменты для REST

- *Postman*. Удобный инструмент для тестирования конечных точек REST. С его помощью вы можете проверять примеры настоящей книги, где создаются конечные точки REST. Однако Postman – это нечто гораздо большее, чем

просто средство тестирования; данный инструмент включает в себя и другие функции, в том числе сценарии автоматизации и средства документирования приложения. Подробнее о Postman читайте на официальной странице: <https://www.postman.com/>.

- *cURL*. Простая утилита командной строки для вызова конечных точек REST. Можно применять как упрощенную альтернативу Postman для тестирования примеров этой книги, где используются конечные точки REST. Загрузить cURL и получить более подробную информацию об ее установке можно здесь: <https://curl.se/download.html>.

MySQL

- *MySQL Server*. СУБД, которая легко устанавливается на местном компьютере для тестирования примеров, требующих доступа к локальной базе данных. Загрузите MySQL Server и почитайте больше об этой СУБД на ее веб-странице: <https://dev.mysql.com/downloads/mysql/>.
- *MySQL Workbench*. Клиент для MySQL Server. С его помощью можно получить доступ к базам данных, находящихся под управлением MySQL Server, чтобы убедиться в корректности работы приложения с сохраненными данными. Узнать подробнее о MySQL Workbench и загрузить этот инструмент можно здесь: <https://www.mysql.com/products/workbench/>.
- *SQLYog*. Альтернатива MySQL Workbench. Загрузить можно по ссылке: <https://webyog.com/product/sqlyog/>.

PostgreSQL

- *PostgreSQL*. СУБД PostgreSQL является альтернативой MySQL. С ее помощью также можно протестировать примеры этой книги или разработать аналогичные приложения, нуждающиеся в базе данных. Подробнее о PostgreSQL вы узнаете на ее веб-странице: <https://www.postgresql.org/download/>.
- *pgAdmin*. Инструмент администрирования PostgreSQL. Если вы решите использовать PostgreSQL вместо MySQL, чтобы выполнять примеры книги, для управления этой СУБД вам понадобится pgAdmin. Больше сведений о pgAdmin здесь: <https://www.pgadmin.org/>.

Приложение Ж

Материалы, рекомендуемые для дальнейшего изучения Spring

В этом приложении перечислен ряд отличных учебных материалов, с помощью которых я рекомендую вам продолжить освоение Spring после прочтения данной книги.

- *Walls C. Spring in Action, 6th ed.* (Manning, 2021). Вначале вы освежите те знания, которые приобрели, прочитав «Spring по-быстрому», а затем познакомитесь с множеством проектов, входящих в экосистему Spring. Вы найдете здесь интересное описание Spring Security, асинхронной передачи данных, Project Reactor, Rsocket и специфики использования активатора Spring Boot.
- *Spilča L. Spring Security in Action* (Manning, 2020). Приложения для обеспечения безопасности — важнейшая тема, которую необходимо изучить сразу после освоения базы. В этой книге подробно рассматривается использование Spring Security для защиты приложения от всевозможных атак путем правильной реализации механизмов авторизации и аутентификации.
- *Хеклер M. Spring Boot по-быстрому* (Питер, 2022). Spring Boot — один из важнейших проектов экосистемы Spring. В настоящее время большинство команд разработчиков используют его, чтобы упростить реализацию Spring-приложений. Именно поэтому мы применяли Spring Boot в более чем половине глав этой книги. После того как усвоите основы, советую глубже нырнуть в Spring Boot. Я считаю эту книгу отличным ресурсом для разработчиков, изучающих Spring.
- *LongJ. Reactive Spring* (2020). Как показывает опыт моих проектов, использование реактивной концепции при создании веб-приложений имеет ряд важных преимуществ. В своей книге автор обсуждает эти преимущества

и показывает, как правильно строить реактивные Spring-приложения. Советую перейти к ней после того, как прочитаете *Spring in Action* Крейга Уоллса.

- *Tudose C. JUnit in Action* (Manning, 2020). Как уже говорилось в главе 15, приложения необходимо тестировать. В данной книге я затронул основы тестирования Spring-приложений. Но эта тема столь сложна, что заслуживает отдельного издания. Катарин в своей книге подробно описывает тестирование Java-приложений. Советую прочитать ее, чтобы закрепить навыки написания тестов.
- *Болье А. Изучаем SQL*, 3-е изд. В главах 12–14 мы рассмотрели построение уровня хранения данных в Spring-приложениях. Там я использовал SQL-запросы, предполагая, что вы уже знакомы с основами SQL. Если вам нужно освежить знания в этой области, советую данную книгу — в ней подробно описаны все основные технологии SQL, которые понадобятся вам в большинстве приложений.
- *Boyarsky J., Selikoff S. OCP Oracle Certified Professional Java SE 11 Developer Complete Study Guide* (Sybex, 2020). Чтобы начать изучать Spring, необходимо знать основы Java. Но иногда забываются даже самые базовые приемы и элементы синтаксиса. Книга Джин и Скотта по подготовке к экзамену OCP — основная, которую я обычно использую, чтобы вспомнить главные синтаксические конструкции. Я всегда читаю последнюю редакцию этого издания, когда готовлюсь к очередной сертификации по OCP.
- Плейлист Spring Framework на моем канале YouTube. Если вам нравится смотреть видеоуроки и прямые трансляции, приглашаю на мой канал YouTube (youtube.com/c/laurentiuspilca), где мы будем обсуждать темы, связанные с Java. Плейлист материалов по Spring находится здесь: <http://mng.bz/yJQE>. Подпишитесь на канал, чтобы получать уведомления, когда я буду размещать там новые видео или запланирую прямые эфиры.
- Мой блог (laurspilca.com/blog). Кроме канала YouTube, у меня еще есть блог, где я пишу статьи. Советую подписаться на него — там вы найдете множество статей о Java.

Лауренциу Спилкэ
Spring быстро

Перевела с английского Е. Сандицкая

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Питиримов</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>К. Тарасевич</i>
Художественный редактор	<i>Б. Мостапан</i>
Корректоры	<i>О. Андриевич, Е. Павлович</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 10.2022. Наименование: книжная продукция. Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные
профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 02.08.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 36,120. Тираж 800. Заказ 0000.

Марк Хеклер

SPRING BOOT ПО-БЫСТРОМУ



Spring Boot, который скачивают более 75 миллионов раз в месяц, — наиболее широко используемый фреймворк Java. Его удобство и возможности совершили революцию в разработке приложений, от монолитных до микросервисов. Тем не менее простота Spring Boot может привести в замешательство. Что именно разработчику нужно изучить, чтобы сразу же выдавать результат? Это практическое руководство научит вас писать успешные приложения для критически важных задач.

Марк Хеклер из VMware, компании, создавшей Spring, проведет вас по всей архитектуре Spring Boot, охватив такие вопросы, как отладка, тестирование и развертывание. Если вы хотите быстро и эффективно разрабатывать нативные облачные приложения Java или Kotlin на базе Spring Boot с помощью реактивного программирования, создания API и доступа к разнообразным базам данных — эта книга для вас.

КУПИТЬ

Марк Лой, Патрик Нимайер, Дэн Лук

ПРОГРАММИРУЕМ НА JAVA

5-е международное издание



Неважно кто вы — разработчик ПО или пользователь, в любом случае слышали о языке Java. В этой книге вы на конкретных примерах изучите основы Java, API, библиотеки классов, приемы и идиомы программирования. Особое внимание авторы уделяют построению реальных приложений.

Вы освоите средства управления ресурсами и исключениями, а также познакомитесь с новыми возможностями языка, появившимися в последних версиях Java.

КУПИТЬ