

Spring

В ДЕЙСТВИИ

ТРЕТЬЕ ИЗДАНИЕ

Крейг Уоллс



Крейг Уоллс

Spring в действии

Craig Walls

Spring in Action



MANNING
SHELTER ISLAND

Крейг Уоллс

Spring в действии



Москва, 2013

УДК 004Spring
ББК 32.973-018.2
У62

Уоллс К.
У62 Spring в действии. – М.: ДМК Пресс, 2013. – 752 с.: ил.

ISBN 978-5-94074-568-6

Фреймворк Spring Framework – необходимый инструмент для разработчиков приложений на Java.

В книге описана последняя версия Spring 3, который несет в себе новые мощные особенности, такие как язык выражений SpEL, новые аннотации для работы с контейнером IoC и поддержка архитектуры REST. Автор, Крейг Уоллс, обладает особым талантом придумывать весьма интересные примеры, сосредоточенные на особенностях и приемах использования Spring, которые действительно будут полезны читателям.

В русскоязычном переводе добавлены главы из 2-го американского издания, которые автор не включил в 3-е издание «Spring in Action».

Издание предназначено как для начинающих пользователей фреймворка, так и для опытных пользователей Spring, желающих задействовать новые возможности версии 3.0.

УДК 004Spring
ББК 32.973-018.2

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

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-9351-8235-1 (анг.)

Copyright © 2011 by Manning Publications Co.

ISBN 978-5-94074-568-6 (рус.)

© Оформление, перевод
ДМК Пресс, 2013



Содержание

Предисловие к русскому изданию	16
Предисловие.....	18
Благодарности	20
Об этой книге	22
Об иллюстрации на обложке	28
Часть I. Ядро Spring	29
Глава 1. Введение в Spring.....	30
1.1. Упрощение разработки на языке Java	32
1.1.1. Свобода использования POJO.....	33
1.1.2. Внедрение зависимостей.....	35
1.1.3. Применение аспектно-ориентированного программирования.....	40
1.1.4. Устранение шаблонного кода с помощью шаблонов	47
1.2. Контейнер компонентов.....	50
1.2.1. Работа с контекстом приложения	51
1.2.2. Жизненный цикл компонента.....	53
1.3. Обзор возможностей Spring.....	54
1.3.1. Модули Spring	55
1.3.2. Дополнительные возможности Spring	59
1.4. Что нового в Spring	64
1.4.1. Что нового в Spring 2.5?.....	64
1.4.2. Что нового в Spring 3.0?.....	65
1.4.3. Что нового в экосистеме Spring?	66
1.5. В заключение	67
Глава 2. Связывание компонентов.....	69
2.1. Объявление компонентов	70



2.1.1. Подготовка конфигурации Spring	71
2.1.2. Объявление простого компонента.....	73
2.1.3. Внедрение через конструкторы.....	74
2.1.4. Область действия компонента.....	80
2.1.5. Инициализация и уничтожение компонентов.....	82
2.2. Внедрение в свойства компонентов.....	84
2.2.1. Внедрение простых значений.....	86
2.2.2. Внедрение ссылок на другие компоненты	87
2.2.3. Связывание свойств с помощью пространства имен р	91
2.2.4. Внедрение коллекций.....	92
2.2.5. Внедрение пустого значения.....	99
2.3. Внедрение выражений.....	99
2.3.1. Основы языка выражений SpEL	100
2.3.2. Выполнение операций со значениями SpEL	105
2.3.3. Обработка коллекций на языке SpEL	111
2.4. В заключение	116

Глава 3. Дополнительные способы связывания компонентов 117

3.1. Объявление родителей и потомков компонентов	118
3.1.1. Абстрактные компоненты	119
3.1.2. Общие абстрактные свойства.....	122
3.2. Внедрение методов	124
3.2.1. Основы замещения методов	125
3.2.2. Использование внедрения методов чтения	130
3.3. Внедрение не-Spring компонентов	132
3.4. Пользовательские редакторы свойств	135
3.5. Специальные компоненты Spring	140
3.5.1. Компоненты постобработки	140
3.5.2. Постобработка контейнера	144
3.5.3. Внешние файлы с настройками свойств	145
3.5.4. Извлечение текстовых сообщений	147
3.5.5. Уменьшение связности с использованием событий ...	150
3.5.6. Создание «осведомленных» компонентов	153
3.6. Компоненты, управляемые сценариями.....	156
3.6.1. Добавляем лайм в кокос.....	158

3.6.2. Компонент, управляемый сценарием.....	159
3.6.3. Внедрение в свойства компонентов, управляемых сценариями	162
3.6.4. Обновление компонентов, управляемых сценариями	164
3.6.5. Создание компонентов, управляемых сценариями, непосредственно в конфигурации.....	165
3.7. В заключение	166

Глава 4. Сокращение размера XML-конфигурации Spring

4.1. Автоматическое связывание свойств компонентов	169
4.1.1. Четыре типа автоматического связывания	169
4.1.2. Смешивание автоматического и явного связывания	175
4.2. Связывание посредством аннотаций	177
4.2.1. Использование аннотации @Autowired	178
4.2.2. Автоматическое связывание с применением стандартной аннотации @Inject.....	183
4.2.3. Использование выражений в аннотациях внедрения зависимостей.....	186
4.3. Автоматическое определение компонентов	188
4.3.1. Аннотирование компонентов для автоматического определения	189
4.3.2. Включение фильтров в элемент component-scans ...	190
4.4. Конфигурирование Spring в программном коде на Java....	192
4.4.1. Подготовка к конфигурированию на языке Java	192
4.4.2. Определение конфигурационных классов	193
4.4.3. Объявление простого компонента.....	194
4.4.4. Внедрение зависимостей в конфигурации на языке Java	195
4.5. В заключение	196

Глава 5. Аспектно-ориентированный Spring

5.1. Знакомство с AOP	200
5.1.1. Определение терминологии AOP	201
5.1.2. Поддержка AOP в Spring	204



5.2. Выбор точек сопряжения в описаниях срезов	207
5.2.1. Определение срезов множества точек сопряжения	209
5.2.2. Использование указателя bean()	210
5.3. Объявление аспектов в XML.....	211
5.3.1. Объявление советов, выполняемых до или после....	213
5.3.2. Объявление советов, выполняемых и до, и после	216
5.3.3. Передача параметров советам.....	218
5.3.4. Внедрение новых возможностей с помощью аспектов	221
5.4. Аннотирование аспектов.....	224
5.4.1. Создание советов, выполняемых и до, и после.....	227
5.4.2. Передача аргументов аннотированным советам	228
5.4.3. Внедрение с помощью аннотаций	230
5.5. Внедрение аспектов AspectJ.....	231
5.6. В заключение	235
Часть II. Основы приложений Spring	237
Глава 6. Работа с базами данных	238
6.1. Философия доступа к данным в Spring.....	239
6.1.1. Знакомство с иерархией исключений доступа к данным в Spring	241
6.1.2. Шаблоны доступа к данным.....	244
6.1.3. Использование классов поддержки DAO	247
6.2. Настройка источника данных	249
6.2.1. Использование источников данных из JNDI	249
6.2.2. Использование пулов соединений.....	250
6.2.3. Источник данных JDBC	252
6.3. Использование JDBC совместно со Spring	253
6.3.1. Борьба с разбуханием JDBC-кода	254
6.3.2. Работа с шаблонами JDBC	258
6.4. Интеграция Hibernate и Spring.....	265
6.4.1. Обзор Hibernate.....	267
6.4.2. Объявление фабрики сеансов Hibernate	268
6.4.3. Создание классов для работы с Hibernate, независимых от Spring	271

6.5. Spring и Java Persistence API	273
6.5.1. Настройка фабрики диспетчера сущностей.....	274
6.5.2. Объект DAO на основе JPA.....	280
6.6. Кеширование.....	282
6.6.1. Настройка кеширования.....	285
6.6.2. Настройка компонентов для кеширования.....	289
6.6.3. Декларативное кеширование с помощью аннотаций	292
6.7. В заключение.....	294
Глава 7. Управление транзакциями.....	296
7.1. Знакомство с транзакциями	297
7.1.1. Описание транзакций в четырех словах.....	299
7.1.2. Знакомство с поддержкой транзакций в Spring	300
7.2. Выбор диспетчера транзакций.....	301
7.2.1. Транзакции JDBC	303
7.2.2. Транзакции Hibernate	303
7.2.3. Транзакции Java Persistence API	304
7.2.4. Транзакции Java Transaction API	305
7.3. Программное управление транзакциями в Spring	306
7.4. Декларативное управление транзакциями.....	309
7.4.1. Определение атрибутов транзакций.....	309
7.4.2. Объявление транзакций в XML.....	315
7.4.3. Определение транзакций с помощью аннотаций.....	318
7.5. В заключение.....	320
Глава 8. Создание веб-приложений с помощью Spring MVC	321
8.1. Обзор Spring MVC	322
8.1.1. Путь одного запроса через Spring MVC.....	322
8.1.2. Настройка Spring MVC	324
8.2. Создание простого контроллера.....	327
8.2.1. Настройка поддержки аннотаций в Spring MVC.....	328
8.2.2. Контроллер главной страницы.....	329
8.2.3. Поиск представлений	334
8.2.4. Объявление представления главной страницы	340
8.2.5. Завершение определения контекста приложения Spring	342



8.3. Контроллер обработки входных данных	344
8.3.1. Создание контроллера, обрабатывающего входные данные.....	345
8.3.2. Представление, отображающее список сообщений	348
8.4. Обработка форм	349
8.4.1. Отображение формы регистрации	350
8.4.2. Обработка данных формы	353
8.4.3. Проверка входных данных	355
8.5. Выгрузка файлов	361
8.5.1. Добавление в форму поля выгрузки файла	361
8.5.2. Прием выгружаемых файлов	363
8.5.3. Настройка Spring для выгрузки файлов	367
8.6. Альтернативы JSP1	368
8.6.1. Использование шаблонов Velocity	368
8.6.2. Использование шаблонов FreeMarker.....	377
8.7. Генерирование вывода, отличного от HTML.....	383
8.7.1. Создание электронных таблиц Excel	384
8.7.2. Создание документов PDF	388
8.8. В заключение.....	391
Глава 9. Использование Spring Web Flow	392
9.1. Установка Spring Web Flow	393
9.1.1. Настройка расширения Web Flow в Spring.....	394
9.2. Элементы последовательности операций.....	397
9.2.1. Состояния	398
9.2.2. Переходы	402
9.2.3. Данные в последовательностях	404
9.3. Соединяем все вместе: последовательность pizza.....	407
9.3.1. Определение основной последовательности	407
9.3.2. Сбор информации о клиенте	412
9.3.3. Оформление заказа	420
9.3.4. Прием оплаты	424
9.4. Безопасность веб-последовательностей	427
9.5. Интеграция Spring Web Flow с другими фреймворками.....	427
9.5.1. JavaServer Faces.....	428
9.6. В заключение	430

Глава 10. Безопасность в Spring	432
10.1. Введение в Spring Security	433
10.1.1. Обзор Spring Security.....	434
10.1.2. Использование конфигурационного пространства имен Spring Security	435
10.2. Безопасность веб-запросов.....	436
10.2.1. Сервлет-фильтры	437
10.2.2. Минимальная настройка безопасности	438
10.2.3. Перехват запросов	443
10.3. Безопасность на уровне представлений	447
10.3.1. Доступ к информации об аутентификации.....	448
10.3.2. Отображение с учетом привилегий.....	449
10.4. Аутентификация пользователей.....	452
10.4.1. Настройка репозитория в памяти	453
10.4.2. Аутентификация с использованием базы данных...	455
10.4.3. Аутентификация с использованием LDAP	457
10.4.4. Включение функции «запомнить меня»	462
10.5. Защита методов	463
10.5.1. Защита методов с помощью аннотации @Secured	464
10.5.2. Использование аннотации JSR-250	
@RolesAllowed	465
10.5.3. Защита с помощью аннотаций, выполняемых до и после вызова	466
10.5.4. Объявление точек внедрения для защиты методов	472
10.6. В заключение	472
Часть III. Интеграция Spring.....	474
Глава 11. Взаимодействие с удаленными службами	476
11.1. Обзор механизмов удаленных взаимодействий в Spring	477
11.2. Использование RMI.....	481
11.2.1. Экспортирование службы RMI	481



11.2.2. Внедрение службы RMI	484
11.3. Экспортирование удаленных служб с помощью Hessian и Burlap.....	488
11.3.1. Экспортирование службы с помощью Hessian/Burlap	489
11.3.2. Доступ к службам Hessian/Burlap	492
11.4. Использование Spring Http Invoker	494
11.4.1. Экспортирование компонентов в виде служб HTTP Invoker	495
11.4.2. Доступ к службам HTTP Invoker	497
11.5. Экспортирование и использование веб-служб	498
11.5.1. Создание конечных точек JAX-WS с поддержкой Spring	500
11.5.2. Проксирование служб JAX-WS на стороне клиента.....	505
11.6. В заключение	507
Глава 12. Поддержка архитектуры REST в Spring	509
12.1. Обзор архитектуры REST	510
12.1.1. Основы REST	510
12.1.2. Поддержка REST в Spring	511
12.2. Создание контроллеров, ориентированных на ресурсы	512
12.2.1. Устройство контроллера, противоречащего архитектуре REST	512
12.2.2. Обработка адресов URL в архитектуре RESTful	514
12.2.3. Выполнение операций в стиле REST	519
12.3. Представление ресурсов	523
12.3.1. Договоренность о представлении ресурса	524
12.3.2. Преобразование HTTP-сообщений	528
12.4. Клиенты REST	532
12.4.1. Операции класса RestTemplate	534
12.4.2. Чтение ресурсов.....	536
12.4.3. Изменение ресурсов	540
12.4.4. Удаление ресурсов	542
12.4.5. Создание новых ресурсов	543
12.4.6. Обмен ресурсами.....	546

12.5. Отправка форм в стиле RESTful	549
12.5.1. Отображение скрытого поля с именем метода	550
12.5.2. Преобразование типа запроса	552
12.6. В заключение	553

Глава 13. Обмен сообщениями в Spring 555

13.1. Краткое введение в JMS	556
13.1.1. Архитектура JMS	557
13.1.2. Преимущества JMS	561
13.2. Настройка брокера сообщений в Spring	563
13.2.1. Создание фабрики соединений	563
13.2.2. Объявление приемников ActiveMQ	565
13.3. Работа с шаблонами JMS в Spring	566
13.3.1. Борьба с разбуханием JMS-кода	566
13.3.2. Работа с шаблонами JMS	568
13.4. Создание POJO, управляемых сообщениями	575
13.4.1. Создание объекта для чтения сообщений.....	576
13.4.2. Настройка обработчиков сообщений.....	578
13.5. Механизмы RPC, основанные на сообщениях	579
13.5.1. Механизм RPC, основанный на сообщениях, в фреймворке Spring	580
13.5.2. Механизм RPC, основанный на сообщениях, в Lingo	583
13.6. В заключение	586

Глава 14. Управление компонентами Spring с помощью JMX 588

14.1. Экспортирование компонентов Spring как управляемых компонентов	589
14.1.1. Экспортирование методов по их именам.....	593
14.1.2. Определение экспортруемых операций и атрибутов с помощью интерфейсов	596
14.1.3. Объявление управляемых компонентов с помощью аннотаций	597
14.1.4. Разрешение конфликтов между управляемыми компонентами	599
14.2. Удаленные компоненты MBean	601



14.2.1. Экспортирование удаленного компонента MBean	602
14.2.2. Доступ к удаленным компонентам MBean.....	603
14.2.3. Проксирование управляемых компонентов	605
14.3. Обработка извещений	606
14.3.1. Прием извещений	609
14.4. В заключение	610
 Глава 15. Создание веб-служб на основе модели contract-first	 611
15.1. Введение в Spring-WS	613
15.2. Определение API службы (в первую очередь!)......	616
15.2.1. Создание примеров XML-сообщений	616
15.3. Обработка сообщений в веб-службе	623
15.3.1. Создание конечной точки на основе модели JDOM	625
15.3.2. Маршалинг содержимого сообщений.....	628
15.4. Связываем все вместе	632
15.4.1. Spring-WS: общая картина	633
15.4.2. Отображение сообщений в конечные точки	634
15.4.3. Настройка конечной точки службы.....	635
15.4.4. Настройка маршалера сообщений	636
15.4.5. Обработка исключений в конечной точке	639
15.4.6. Создание WSDL-файлов	641
15.4.7. Разворачивание службы.....	646
15.5. Использование веб-служб Spring-WS	647
15.5.1. Работа с шаблонами веб-служб	648
15.5.2. Использование поддержки шлюза веб-служб	656
15.6. В заключение	658
 Глава 16. Spring и Enterprise JavaBeans	 660
16.1. Внедрение компонентов EJB в Spring.....	661
16.1.1. Проксирование сеансовых компонентов (EJB 2.x).....	663
16.1.2. Внедрение компонентов EJB в компоненты Spring	668
16.2. Разработка компонентов с поддержкой Spring (EJB 2.x)	670

16.3. Spring и EJB3	673
16.3.1. Pitchfork.....	674
16.3.2. Введение в Pitchfork	676
16.3.3. Внедрение ресурсов с помощью аннотации	676
16.3.4. Объявление перехватчиков с помощью аннотаций.....	678
16.4. В заключение	680
Глава 17. Прочее	682
17.1. Импортирование внешних настроек.....	683
17.1.1. Подстановка переменных-заполнителей	684
17.1.2. Переопределение свойств	687
17.1.3. Шифрование внешних определений свойств.....	689
17.2. Внедрение объектов из JNDI	692
17.2.1. Работа с обычным JNDI API.....	692
17.2.2. Внедрение объектов из JNDI	695
17.2.3. Внедрение компонентов EJB в Spring	699
17.3. Отправка электронной почты	701
17.3.1. Настройка отправки электронной почты	701
17.3.2. Создание электронных писем.....	704
17.4. Выполнение заданий по расписанию и в фоновом режиме	711
17.4.1. Объявление методов, вызываемых по расписанию	712
17.4.2. Объявление асинхронных методов	714
17.5. В заключение	716
17.6. Конец?	717
Предметный указатель.....	719



Предисловие к русскому изданию

Русское издание книги «Spring в действии» представляет собой объединение 2-го и 3-го изданий оригинальной книги «Spring in Action». Автор, пытаясь максимально сократить объем нового издания, убрал из него обсуждение тем, достаточно важных на наш взгляд, и постоянно отсылает интересующихся к предыдущему изданию. По этой причине было принято решение добавить в перевод нового издания главы и разделы из предыдущего оригинального издания (перечисленные ниже), где обсуждаются темы, остающиеся актуальными и поныне.

Глава 3 «Дополнительные способы связывания компонентов» (глава 3 во 2-м издании) рассматривает дополнительные приемы связывания компонентов. Эти приемы не имеют такого же широкого применения, как приемы связывания, представленные в главе 2, но не теряют от этого своей практической ценности.

Раздел 6.6 «Кеширование» (раздел 5.7 во втором издании) демонстрирует приемы кеширования, играющие важную роль в обеспечении высокой производительности приложений.

Раздел 9.5 «Интеграция Spring Web Flow с другими фреймворками» (раздел 15.4 во втором издании). В 3-м издании демонстрируются приемы использования фреймворка Spring Web Flow, однако автор убрал из третьего издания разделы, демонстрирующие возможность интеграции Spring Web Flow с другими фреймворками, такими как Jakarta Struts и JavaServer Faces. Если интеграция с фреймворком Jakarta Struts действительно потеряла свою актуальность, то проблема интеграции с JavaServer Faces все еще представляет интерес для разработчиков.

Глава 15 «Создание веб-служб на основе модели contract-first» демонстрирует применение фреймворка Spring-WS для создания веб-служб с применением модели «contract-first», когда сначала создается WSDL-определение веб-службы, а затем на его основе разрабатывается реализация, за счет чего достигается отделение внешнего и внутреннего API службы.

Глава 16 «Spring и Enterprise JavaBeans» описывает приемы внедрения компонентов EJB в контекст Spring, способы создания компонентов EJB с поддержкой Spring и возможность применения аннотаций EJB 3 к компонентам Spring.

К сожалению, во 2-м и 3-м изданиях обсуждение основывается на разных примерах, поэтому в перечисленных выше главах и разделах (кроме главы 3) читатель столкнется с упоминанием примеров, не обсуждавшихся ранее. Мы искренне надеемся, что это не помешает читателю ухватить основной смысл и заранее приносим свои извинения за несогласованность.



Предисловие

Ого! Прошло почти семь лет с того момента, как была выпущена версия Spring 1.0 и мы с Райаном Брейденбахом (Ryan Breidenbach) приступили к работе над первым изданием книги «Spring в действии». Кто бы мог предположить тогда, что Spring окажет такое сильное влияние на разработку приложений на языке Java?

В первом издании Райан и я пытались охватить все аспекты фреймворка Spring. И мы во многом преуспели. В то время все, что касалось Spring, легко можно было уместить в 11 глав, вместе с описанием приемов внедрения зависимостей, АОР, хранения данных, транзакций, Spring MVC и Acegi Security в главных ролях. Конечно, тогда все это необходимо было сдобрить большим количеством XML. (Помнит ли кто-нибудь, как выглядело объявление транзакций с применением `TransactionProxyFactoryBean`?)

К тому времени, когда я взялся за второе издание книги, фреймворк Spring значительно вырос. Я снова попытался уместить все в единственную книгу и обнаружил, что это невозможно. Фреймворк Spring вырос настолько, что его невозможно стало описать в 700–800-страничной книге. Из второго издания фактически пришлось вырезать целые главы из-за нехватки места.

С момента выхода второго издания прошло три года, и сменились две основные версии Spring. Фреймворк Spring разросся еще больше, и теперь пришлось бы написать несколько томов, чтобы полностью охватить все его возможности. Уместить все необходимые знания о Spring в одну книгу стало просто нереально.

Поэтому я даже не пытался сделать это.

Часто с каждым последующим изданием книги становятся все толще. Но вы наверняка обратили внимание, что это, третье издание «Spring в действии», содержит меньше страниц, чем второе издание. Это стало возможным по двум причинам.

Так как я не смог уместить все в один том, мне пришлось выбирать, какие темы должны быть включены. Я решил сосредоточиться на самых основных, которые, на мой взгляд, должно знать

большинство разработчиков, использующих фреймворк Spring. Это не говорит о том, что другие темы неважны, но эти составляют самую суть разработки с применением Spring.

Другая причина, почему это издание стало меньше, в том, что одновременно с ростом фреймворк Spring становился проще с каждым выпуском. Богатый набор пространств имен, конфигураций Spring, заимствование моделей программирования на основе аннотаций, а также внедрение рациональных соглашений и умолчаний уменьшили конфигурацию Spring, которая превратилась из последовательности страниц разметки XML в горстку элементов.

Но будьте уверены: несмотря на меньшее число страниц, мне удалось уместить в них много новых сведений о Spring. Наряду с описанием приемов внедрения зависимостей, AOP и декларативных транзакций, существующих уже относительно давно, книга содержит новые сведения, появившиеся или изменившиеся после выхода второго издания, часть которых перечислена ниже:

- ❑ создание компонентов на основе аннотаций существенно уменьшает размер XML-конфигурации Spring;
- ❑ новый язык выражений для динамического вычисления значений свойств компонентов во время выполнения;
- ❑ совершенно новый фреймворк Spring MVC, действующий на основе аннотаций, который стал намного гибче прежнего иерархического фреймворка контроллеров;
- ❑ обеспечение безопасности приложений с помощью фреймворка Spring Security стало намного проще благодаря новому конфигурационному пространству имен, удобным значениям по умолчанию и поддержке правил безопасности, ориентированных на применение выражений;
- ❑ отличная поддержка конструирования и обработки ресурсов REST, основанная на фреймворке Spring MVC.

Надеюсь, что эта книга станет исчерпывающим руководством и для опытных, и для начинающих разработчиков, использующих фреймворк Spring в своих проектах.



Благодарности

Прежде чем вы смогли взять эту книгу в руки, она прошла через руки многих других людей – людей, которые редактировали ее, рецензировали, корректировали и управляли всем процессом публикации. Вы не смогли бы сейчас читать эту книгу, если бы не все эти люди.

Во-первых, я хотел бы поблагодарить всех сотрудников издательства Manning за их упорный труд, за то, что подталкивали меня, чтобы я наконец-то закончил эту книгу, и за то, что сделали эту книгу лучше, чем она могла бы быть: Мариан Бейс (Marjan Bace), Майкл Стефенс (Michael Stephens), Кристина Рудольф (Christina Rudloff), Карен Тегтмейер (Karen Tegtmeyer), Морин Спенсер (Maureen Spencer), Мэри Пиргис (Mary Piergies), Себастьян Стирлинг (Sebastian Stirling), Бенджамин Берг (Benjamin Berg), Кэти Теннант (Katie Tennant), Жанет Вейл (Janet Vail) и Дотти Мариско (Dottie Marsico).

Попутно несколько человек получили возможность прочитать рукопись в самом сыром виде и откликнулись, сообщив, где я был прав, а где (со вздохом) промахнулся. Большое спасибо всем рецензентам за их ценные отклики: Валентин Греттаз (Valentin Crettaz), Джейф Эдисон (Jeff Addison), Джон Райан (John Ryan), Оливер Нугье (Olivier Nouguier), Джошуа Уайт (Joshua White), Диевихан Наллажагаппан (Deiveehan Nallazhagappan), Адам Тафт (Adam Taft), Питер Павлович (Peter Pavlovich), Микель Элвис (Mykel Alvis), Рик Вагнер (Rick Wagner), Патрик Стегер (Patrick Steger), Джош Девинс (Josh Devins), Дэн Алфорд (Dan Alford), Альберто Лагна (Alberto Lagna), Дэн Добрин (Dan Dobrin), Роберт Хансон (Robert Hanson), Чад Дэвис (Chad Davis), Кэрол Макдональд (Carol McDonald), Дипак Вохра (Deepak Vohra) и Роберт О'Коннор (Robert O'Connor). Особое спасибо Дугу Уоррену (Doug Warren), взявшему на себя роль технического рецензента и причесавшего описание технических деталей частой гребенкой.

Также хотелось бы выразить благодарность всем тем, кто не принимал прямого участия в создании книги, но оказывал дружескую

поддержку, дарил доброе общение и способствовал тому, чтобы я делал перерывы в работе над книгой и занимался другими делами.

В первую очередь я благодарен моей супруге Рейми (Raumie). Ты – мой лучший друг, любовь моей жизни и вдохновитель всех моих свершений. Я очень люблю тебя. Спасибо за помощь во время работы над книгой и поддержку.

Спасибо вам, мои маленькие принцессы Мейси (Maisy) и Мади (Madi), за ваши объятия, смех, воображение и случайные перерывы для игры в Mario Kart.

Спасибо моим коллегам по SpringSource, что продолжают ме-нять взгляды на разработку программного обеспечения и за предо-ставленную мне возможность быть частью нашей команды. Особое спасибо двум моим коллегам, с которыми я работаю каждый день, Кейту Дональду (Keith Donald) и Рою Кларксону (Roy Clarkson) – мы много удивительного сделали в прошлом году, и я с нетерпением жду немало удивительного в будущем.

Большое спасибо моим соратникам по конференции «No Fluff/Just Stuff», которые раз в несколько выходных напоминают мне, что я не так умен, как они: Тед Ньюард (Ted Neward), Венкат Сабраманиам (Venkat Subramaniam), Тим Берглунд (Tim Berglund), Мэттью Маккалло (Matthew McCullough), Мэтт Стин (Matt Stine), Брайан Готц (Brian Goetz), Джейф Браун (Jeff Brown), Дэйв Клейн (Dave Klein), Кен Сип (Ken Sipe), Наталия Шатт (Nathaniel Schutta), Нил Форд (Neal Ford), Патрик Пател (Pratik Patel), Рохит Бхардвай (Rohit Bhardwaj), Скотт Дэвис (Scott Davis), Марк Ричардс (Mark Richards) и конечно же Джей Циммерман (Jay Zimmerman).

Наконец, существует множество других людей, кому я хотел бы послать слова благодарности за их участие в формировании меня, в моей карьере и этой книге: Райан Брейденбах (Ryan Breidenbach), Бен Ради (Ben Rady), Майк Нэш (Mike Nash), Мэтт Смит (Matt Smith), Джон Вудвард (John Woodward), Грэг Вон (Greg Vaughn), Барри Роджерс (Barry Rogers), Пол Хользер (Paul Holser), Дерек Лэн (Derek Lane), Эрик Вейбуст (Erik Weibust) и Эндрю Рубаль-каба (Andrew Rubalcaba).



Об этой книге

Фреймворк Spring Framework создавался с весьма конкретной целью – чтобы упростить разработку приложений Java EE. Аналогично и книга «Spring в действии» была написана, чтобы упростить обучение применению фреймворка Spring. Моя цель не в том, чтобы дать вам подробный перечень Spring API. Вместо этого я надеюсь описать Spring Framework как разработчик приложений Java EE, представив практические примеры программного кода. Поскольку фреймворк Spring имеет модульную организацию, эта книга организована таким же образом. Я понимаю, что разные разработчики имеют разные потребности. Некоторые могут желать изучать Spring Framework, начиная с самых основ, другие – проявлять разборчивость в выборе тем и двигаться в собственном темпе. В этом смысле начинающие могут воспринимать книгу как инструмент для первоначального знакомства с фреймворком Spring, а желающие глубже вникнуть в конкретные особенности – как руководство и справочник.

Кому адресована эта книга

Третье издание книги «Spring в действии» адресовано всем Java-разработчикам, но особенно полезной она будет для разработчиков корпоративных приложений на Java. Я проведу вас через примеры программного кода, сложность которых будет расти на протяжении каждой главы, но вы должны понимать, что истинная мощь фреймворка Spring проявляется в простоте разработки крупных приложений. Поэтому разработчики корпоративных приложений смогут в полной мере оценить примеры, представленные в книге.

Большая часть фреймворка Spring обеспечивает реализацию корпоративных служб, между Spring и EJB можно провести множество параллелей. Поэтому в сравнении этих двух фреймворков будет полезен любой опыт. Данной теме будет посвящена часть этой книги. Фактически последние пять глав демонстрируют способности фреймворка Spring поддерживать интеграцию веб-приложений.

Особенно ценной эту часть книги найдут разработчики корпоративных приложений.

Структура книги

Книга «Spring в действии» делится на три части. Первая часть является введением в основы фреймворка Spring Framework. Во второй части будут представлены общие элементы приложений на основе фреймворка Spring. Заключительная часть покажет, как можно использовать фреймворк Spring для интеграции с другими приложениями и службами.

В первой части будет проведено исследование двух основных особенностей фреймворка Spring Framework – внедрения зависимостей (Dependency Injection, DI) и аспектно-ориентированного программирования (Aspect-Oriented Programming, AOP). Это даст возможность хорошо усвоить основы Spring, которые будут использоваться на протяжении всей книги.

В главе 1 будут представлены DI и AOP и рассказано, как их использовать в разработке слабосвязанных Java-приложений.

В главе 2 подробно будет рассказываться, как конфигурировать и связывать прикладные объекты, используя прием внедрения зависимостей. Она покажет, как писать слабосвязанные компоненты и устанавливать их зависимости и свойства в контейнере Spring с использованием XML.

После овладения основами связывания компонентов можно приступить к знакомству с дополнительными особенностями контейнера Spring в главе 3. Среди прочего здесь рассказывается, как управлять жизненным циклом компонентов приложения, как определять отношения родитель/потомок в конфигурациях компонентов и как внедрять компоненты сценариев, написанных на языках Ruby и Groovy.

После знакомства в главе 4 с основами конфигурирования Spring с помощью XML будут представлены аспектно-ориентированные альтернативы.

Глава 5 рассматривает возможность использования Spring AOP для отделения сквозных задач от объектов, которые они обслуживают. Эта глава также готовит почву для последующих глав, где Spring AOP будет использоваться для предоставления декларативных услуг, таких как транзакции, безопасность и кеширование.

Вторая часть основывается на особенностях DI и AOP, представленных в первой части, и показывает, как использовать эти понятия для создания общих элементов приложения.



Глава 6 охватывает механизм хранения данных в Spring. Здесь будет представлена поддержка JDBC в Spring, позволяющая убрать из приложения массу шаблонного программного кода, связанного с использованием JDBC. Здесь также будет показано, как Spring взаимодействует с фреймворками хранения данных, такими как Hibernate и Java Persistence API (JPA).

Глава 7 дополняет главу 6, демонстрируя возможность обеспечения целостности баз данных с использованием поддержки транзакций в Spring. Она покажет, как Spring с помощью АОР позволяет наделить простые прикладные объекты мощью декларативных транзакций.

В главе 8 будет представлен веб-фреймворк MVC, входящий в состав Spring. Здесь будет показано, как Spring может прозрачно связывать веб-параметры бизнес-объектов и одновременно обеспечивать контроль допустимости значений и обработку ошибок. Здесь также будет показано, насколько просто расширять функциональность веб-приложений с помощью контроллеров Spring MVC.

В главе 9 исследуется Spring Web Flow – расширение Spring MVC, позволяющее создавать диалоговые веб-приложения. В этой главе рассказывается, как создавать веб-приложения, обеспечивающие выполнение действий пользователями в определенной последовательности.

В главе 10 рассказывается, как обеспечить защиту приложений с помощью Spring Security. Здесь будет показано, как Spring Security защищает приложение на уровне веб-запросов, с использованием сервлетов-фильтров, и на уровне метода, с использованием Spring АОР.

После создания приложения на основе тех знаний, что были получены во второй части, может появиться желание организовать его взаимодействие с другими приложениями или службами. Как это сделать, вы узнаете в третьей части книги.

Глава 11 рассматривает возможность представления прикладных объектов в виде удаленных служб. Здесь также рассказывается, насколько просто можно получить доступ к удаленным службам, как если бы они были обычными объектами в приложении. В число рассматриваемых здесь технологий распределенных вычислений входят RMI, Hessian/Burlap, веб-службы, действующие на основе протокола SOAP, и собственная технология Spring HTTP Invoker.

Глава 12 вновь возвращается к фреймворку Spring MVC и демонстрирует возможность его использования для обеспечения доступа

к данным в приложении как к ресурсам RESTful. Кроме того, здесь рассказывается, как создаются REST-клиенты с помощью Spring RestTemplate.

Глава 13 рассказывает о возможности использования Spring для передачи и получения асинхронных сообщений с помощью JMS. Помимо основных операций, поддерживаемых JMS API в Spring, здесь также будет показано, как использовать открытый проект Lingo для реализации и использования удаленных служб на базе JMS.

Глава 14 покажет, как управлять прикладными объектами с помощью JMX.

В главе 15 будет представлен иной подход к реализации веб-служб, с применением фреймворка Spring-WS. Здесь вы узнаете, как с помощью Spring-WS создавать веб-службы с применением модели «contract-first», когда определение API службы отделено от его реализации.

Несмотря на то, что фреймворк Spring в значительной степени устранил необходимость использовать компоненты EJB, на практике все же может возникнуть потребность совместного использования Spring и EJB. Возможности интеграции Spring и EJB исследуются в главе 16. Здесь вы узнаете, как писать компоненты EJB с поддержкой Spring, как внедрять ссылки на компоненты EJB в контекст приложения Spring и даже как использовать EJB-подобные аннотации для настройки компонентов Spring.

В заключение исследования фреймворка Spring, в главе 17, будет показано, как задействовать Spring для выполнения заданий по расписанию, отправлять электронную почту, обращаться к ресурсам, настроенным с использованием JNDI.

Соглашения об оформлении программного кода

Книга содержит множество примеров программного кода. Эти примеры всегда будут напечатаны моноширинным шрифтом. Если мне потребуется привлечь ваше внимание к какой-то части примера, он будет выделен жирным шрифтом. Все имена классов, методов и фрагменты XML в тексте также будут напечатаны моноширинным шрифтом.

Многие классы и пакеты в Spring имеют весьма длинные (но выразительные) имена. Вследствие этого везде, где необходимо, будут добавляться стрелки (⇒), обозначающие продолжение строки.

Не все примеры программного кода в этой книге будут полными. Часто я буду показывать только один-два метода класса, о котором



идет речь, чтобы не уходить в сторону от обсуждаемой темы. Полные исходные тексты приложений, представленных в книге, можно загрузить с веб-сайта издательства www.manning.com/SpringinActionThirdEdition.

Об авторе

Крейг Уоллс – программист с 13-летним опытом и соавтор книги «XDoclet in Action» (Manning, 2003) и двух предыдущих изданий «Spring in Action» (Manning, 2005 и 2007). Он горячо пропагандирует фреймворк Spring Framework, часто выступая перед местными группами пользователей и на конференциях, а также ведет обсуждение Spring в своем блоге. Когда Крейг не пишет программный код, он проводит все свое свободное время с супругой и двумя дочерьми, шестью птицами, четырьмя собаками, двумя котами и постоянно меняющимся числом тропических рыб. Крейг живет в Плано, штат Техас.

Автор в сети

Одновременно с покупкой третьего издания книги «Spring в действии» вы получаете бесплатный доступ к частному веб-форуму, организованному издательством Manning Publications, где можно оставлять комментарии о книге, задавать технические вопросы, а также получать помощь от автора и других пользователей. Чтобы получить доступ к форуму и зарегистрироваться на нем, откройте в веб-браузере страницу www.manning.com/SpringinActionThirdEdition. Здесь описывается, как попасть на форум после регистрации, какие виды помощи доступны и правила поведения на форуме.

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

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

О названии

Сочетая в себе введение, краткий обзор и примеры использования, книги из серии «в действии» предназначены для изучения и

запоминания. Согласно исследованиям в когнитивистике, люди запоминают вещи, которые они узнают в процессе самостоятельного изучения.

В издательстве Manning нет ученых-когнитивистов, тем не менее мы уверены, что для надежного усваивания необходимо пройти через стадии исследования, игры и, что интересно, пересказа всего, что было изучено. Люди усваивают новые знания до уровня овладения ими, только после их активного исследования. Люди учатся в действии. Особенность учебников из серии «в действии» – в том, что они основываются на примерах. Это побуждает читателя пробовать, играть с новым кодом, исследовать новые идеи.

Есть и другая, более прозаическая причина выбора такого названия книги: наши читатели – занятые люди. Они используют книги для выполнения работы или для решения проблем. Им нужны книги, позволяющие легко перепрыгивать с места на место и изучать только то, что они хотят, и только когда они этого хотят. Им нужны книги, помогающие в действии. Книги этой серии создаются для таких читателей.



Об иллюстрации на обложке

На обложке третьего издания книги «Spring в действии» изображен «Le Caraco» – житель района Карак на юго-западе Иордана. Столица Иордана – город Эль-Карак, который славится древним замком на холме с завораживающим видом на Мертвое море и окрестности.

Рисунок взят из Французского туристического путеводителя «Encyclopedie des Voyages» (автор J. G. St. Saveur), выпущенного в 1796 году. Путешествия ради удовольствия были сравнительно новым явлением в то время, и туристические справочники, такие как этот, были популярны, они позволяли знакомиться с жителями других регионов Франции и других стран, не слезая с кресла.

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

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

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



ЧАСТЬ I. ЯДРО SPRING

Фреймворк Spring обладает массой функциональных возможностей, однако, если разбить их на составные части, можно выделить две его основные особенности: внедрение зависимостей (DI) и аспектно-ориентированное программирование (AOP). В первой главе «Введение в Spring» будет дан краткий обзор DI и AOP в Spring и показано, как они могут помочь в ослаблении связей между объектами в приложении.

В главе 2 «Связывание компонентов» более детально рассматривается, как обеспечить слабую связь между объектами с помощью приема внедрения зависимостей и используя конфигурацию Spring на основе XML. Здесь рассказывается, как определять прикладные объекты и затем связывать их посредством зависимостей.

Перейдя на ступень выше, в главе 3 «Дополнительные способы связывания компонентов» исследуем некоторые дополнительные особенности контейнера Spring и продемонстрируем некоторые более мощные приемы конфигурирования Spring.

XML – не единственный способ конфигурирования Spring. Продолжая с того места, где закончилась предыдущая глава, глава 4 «Сокращение размера XML-конфигурации Spring» расскажет о некоторых новых особенностях в Spring, позволяющих организовать связывание прикладных объектов с минимальным объемом разметки XML (а в некоторых случаях вообще без нее).

Глава 5 «Аспектно-ориентированный Spring» рассказывает, как использовать аспектно-ориентированные особенности Spring AOP для отделения системных служб (таких как безопасность и аудит) от обслуживаемых ими объектов. Эта глава готовит почву для глав 7 и 10, где рассказывается, как использовать Spring AOP для обеспечения декларативных транзакций и безопасности.



Глава 1. Введение в Spring

В этой главе рассматриваются следующие темы:

- ❑ обзор основных модулей Spring;
- ❑ разделение прикладных объектов;
- ❑ управление сквозными задачами с помощью АОР;
- ❑ контейнер компонентов в Spring.

Все началось с компонента.

В 1996 году язык программирования Java был еще новой, перспективной платформой, вызывающей большой интерес. Многие разработчики пришли в этот язык после того, как они увидели, как с помощью апплетов можно создавать полнофункциональные и динамические веб-приложения. Однако вскоре они увидели, что этот новый и странный язык способен на большее, нежели простое управление мультишарнирными героями. В отличие от других существующих языков, Java позволил писать сложные приложения, состоящие из отдельных частей. Они пришли ради апплетов, а остались ради компонентов.

В декабре 1996 года компания Sun Microsystems опубликовала спецификацию JavaBeans 1.00-А, определившую модель программных компонентов Java – набор приемов программирования, позволяющих повторно использовать простые Java-объекты и легко конструировать из них более сложные приложения. Несмотря на то что компоненты JavaBeans были задуманы как универсальный механизм определения повторно используемых прикладных компонентов, они использовались преимущественно в качестве шаблона для создания элементов пользовательского интерфейса. Они казались слишком простыми для «настоящей» работы. Промышленным программистам требовалось нечто большее.

Сложные приложения часто требуют таких услуг, как поддержка транзакций, безопасность и распределенные вычисления, которые не были предусмотрены спецификацией JavaBeans. Поэтому в марте 1998 года компания Sun Microsystems опубликовала новую спецификацию – Enterprise JavaBeans (EJB) 1.0. Эта спецификация

расширила понятие серверных Java-компонентов, обеспечив столь необходимые службы, однако она не смогла поддержать той простоты, что была заложена в оригинальную спецификацию JavaBeans. Кроме похожего названия, спецификация EJB имеет мало общего со спецификацией JavaBeans.

Несмотря на появление множества успешных приложений, созданных на основе спецификации EJB, она никогда не применялась по прямому назначению: для упрощения разработки корпоративных приложений. Это правда, что декларативная модель программирования EJB упрощает многие инфраструктурные аспекты разработки, такие как транзакции и безопасность. Но она привнесла другие сложности, требуя создания дескрипторов развертывания и использования шаблонного кода (домашние и удаленные/локальные интерфейсы). С течением времени многие разработчики разочаровались в EJB, что привело к спаду популярности данной технологии и поиску более простых путей.

Сегодня разработка Java-компонентов вернулась к своим истокам. Новые технологии программирования, включая аспектно-ориентированное программирование (AOP) и внедрение зависимостей (DI), дают JavaBeans дополнительные возможности, ранее заложенные в EJB. Эти технологии оснащают обычные Java-объекты (Plain-Old Java Objects, POJO) моделью декларативного программирования, напоминающей EJB, но без всей сложности спецификации EJB. Больше не надо создавать громоздкий компонент EJB, когда достаточно простого компонента JavaBean.

Следует отметить, что даже EJB эволюционировала к модели программирования на основе POJO. Заимствуя идеи AOP и DI, последняя спецификация EJB стала существенно проще, чем предшествующие. Однако многие разработчики расценили этот шаг как слишком маленький и сделанный достаточно поздно. К моменту выхода спецификации EJB 3 другие фреймворки на основе модели POJO уже зарекомендовали себя в сообществе пользователей Java как фактические стандарты.

Фреймворк Spring Framework, который будет изучаться на протяжении всей этой книги, является передовым средством разработки приложений на основе POJO. В этой главе Spring Framework будет рассматриваться с общей точки зрения, чтобы дать представление о том, что такое Spring. Данная глава позволит получить хорошее понимание круга задач, решаемых фреймворком Spring, и подготовит почву для остальной части книги. Для начала выясним, что такое Spring.



1.1. Упрощение разработки на языке Java

Spring – это свободно распространяемый фреймворк, созданный Родом Джонсоном (Rod Johnson) и описанный в его книге «Expert One-on-One: J2EE Design and Development». Он был создан с целью устраниćи сложности разработки корпоративных приложений и сделать возможным использование простых компонентов JavaBean для достижения всего того, что ранее было возможным только с использованием EJB. Однако область применения Spring не ограничивается разработкой программных компонентов, выполняющихся на стороне сервера. Любое Java-приложение может использовать преимущества фреймворка в плане простоты, тестируемости и слабой связанности.

Термины, обозначающие компоненты. Несмотря на то, что при обсуждении фреймворка Spring для обозначения программных компонентов слова «bean» и «JavaBean» используются как синонимы, это не означает, что компоненты в Spring должны точно соответствовать спецификации JavaBeans. Компоненты в Spring могут быть любыми простыми объектами модели POJO. В этой книге термин «JavaBeans» будет использоваться как синоним термина «POJO» (обычный Java-объект).

Как будет не раз показано на протяжении этой книги, фреймворк Spring обладает весьма широкими возможностями. Но в основе практических всех его особенностей лежат несколько фундаментальных идей, направленных на достижение главной цели – *упрощение разработки приложений на языке Java*.

Весьма смелое заявление! Разработчики многих фреймворков утверждают, что их продукты упрощают те или иные аспекты разработки. Но целью фреймворка Spring является упрощение разработки приложений Java вообще. Это требует некоторых пояснений. Так как же фреймворк Spring упрощает разработку на языке Java?

В своем устремлении на сложности, связанные с разработкой на языке Java, фреймворк Spring использует четыре ключевые стратегии:

- ❑ легковесность и ненасильственность благодаря применению простых Java-объектов (POJO);
- ❑ слабое связывание посредством внедрения зависимостей и ориентированности на интерфейсы;
- ❑ декларативное программирование через аспекты и общепринятые соглашения;
- ❑ уменьшение объема типового кода через аспекты и шаблоны.

Практически все возможности фреймворка Spring уходят корнями в эти стратегии. В остальной части главы каждая из этих идей будет рассматриваться более подробно на конкретных примерах, позволяющих увидеть, как фреймворк Spring действительно упрощает разработку приложений на языке Java. Для начала посмотрим, как он обеспечивает ненасильственность, поощряя использование простых объектов.

1.1.1. Свобода использования POJO

Те, кто имеет опыт достаточно продолжительной разработки на языке Java, вероятно, видели (и даже могли использовать) фреймворки, вынуждающие расширять свои классы или предусматривать реализацию своих интерфейсов. Классическим примером являются сеансовые компоненты эры EJB 2. Как показано в простейшем примере `HelloWorldBean`, спецификация EJB 2 предъявляет достаточно сложные требования:

Листинг 1.1. Спецификация EJB 2.1 вынуждает реализовывать ненужные методы

```
package com.habuma.ejb.session;

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;

public class HelloWorldBean implements SessionBean { // Зачем все эти методы
    public void ejbActivate() {
    }

    public void ejbPassivate() {
    }

    public void ejbRemove() {
    }

    public void setSessionContext(SessionContext ctx) {
    }

    public String sayHello() { // Основная логика компонента EJB
        return "Hello World";
    }

    public void ejbCreate() {
    }
}
```

Интерфейс `SessionBean` обеспечивает возможность управления жизненным циклом компонента EJB за счет реализации различных методов обратного вызова (имена этих методов начинаются с последовательности символов `ejb`). Или, говоря другими словами, интерфейс `SessionBean` вынуждает вторгаться в жизненный цикл компонента EJB, даже если в этом нет необходимости. Масса программного кода в примере `HelloWorldBean` нужна только ради удовлетворения нужд фреймворка. В результате возникает вопрос: кто для кого работает?

Однако спецификация EJB 2 не единственная в своей насыщенности. Другие популярные фреймворки, такие как ранние версии Struts, WebWork и Tapestry, накладывали свой отпечаток на иначе простые Java-классы. Эти тяжеловесные фреймворки вынуждали разработчиков создавать классы, захламленные ненужным программным кодом, часто сложные в тестировании.

Фреймворк Spring не вынуждает (насколько это возможно) захламлять приложения программным кодом для поддержки своего API. Он практически никогда не заставляет обеспечивать реализацию своих интерфейсов или наследовать свои классы. Напротив, в приложениях, основанных на фреймворке Spring, классы часто вообще не имеют никаких признаков, по которым можно было бы судить, что они используются фреймворком. В худшем случае класс может быть аннотирован одной из аннотаций Spring, но во всех остальных отношениях он будет обычным объектом POJO.

Для иллюстрации класс `HelloWorldBean`, представленный в листинге 1.1, можно преобразовать в компонент, управляемый фреймворком Spring, как показано в листинге 1.2.

Листинг 1.2. Фреймворк Spring не выдвигает необоснованных требований к классу `HelloWorldBean`

```
package com.habuma.spring;

public class HelloWorldBean {
    public String sayHello() { // Это все, что необходимо
        return "Hello World";
    }
}
```

Так лучше? Исчезли все ненужные методы управления жизненным циклом. Класс `HelloWorldBean` не реализует, не наследует и не

импортирует ничего из Spring API. Класс `HelloWorldBean` мал, краток и во всех смыслах является простым Java-объектом.

Несмотря на свою простоту, простые Java-объекты могут обладать большими возможностями. Одним из механизмов Spring, увеличивающих мощь простых объектов, является их объединение с помощью внедрения зависимостей. Рассмотрим, как внедрение зависимостей помогает сохранить прикладные объекты независимыми друг от друга.

1.1.2. Внедрение зависимостей

Для кого-то фраза «внедрение зависимостей» может звучать устрашающе, вызывая в воображении сложные приемы программирования или шаблоны проектирования. Однако на самом деле DI не настолько сложно, как кажется. На самом деле применение DI в проектах позволяет существенно упростить программный код, облегчит его понимание и тестирование.

Любое нетривиальное приложение (более сложное, чем простое приложение Hello World) состоит из двух или более классов, которые взаимодействуют друг с другом, реализуя некоторую логику. Обычно каждый объект ответствен за получение собственных ссылок на объекты, с которыми он взаимодействует (его зависимости). Это может привести к сильной связности и сложностям при тестировании.

Например, взгляните на класс **рыцаря** в листинге 1.3 ниже.

Листинг 1.3. Класс `DamselRescuingKnight` может принимать только экземпляр класса `RescueDamselQuests`

```
package com.springinaction.knights;

public class DamselRescuingKnight implements Knight {
    private RescueDamselQuest quest;

    public DamselRescuingKnight() {
        quest = new RescueDamselQuest(); // Тесная связь с классом
    }                                         // RescueDamselQuest

    public void embarkOnQuest() throws QuestException {
        quest.embarke();
    }
}
```

Как показано в листинге 1.3, экземпляр `DamselRescuingKnight` создает собственный экземпляр `RescueDamselQuest` в конструкторе. Это обуславливает тесную связь между `DamselRescuingKnight` и `RescueDamselQuest` и существенно ограничивает возможности рыцаря. Если потребуется спасти даму, он готов будет совершить этот подвиг. Но если потребуется убить дракона или, например, стать рыцарем Круглого стола, он не сможет сделать этого¹.

Но, что хуже всего, для класса `DamselRescuingKnight` очень сложно будет написать модульный тест. В процессе тестирования желательно было бы убедиться, что при вызове метода `embarkOnQuest()` класса `DamselRescuingKnight` вызывается метод `embark()` класса `RescueDamselQuest`. Но в данном случае нет очевидного способа реализовать такую проверку. К сожалению, класс `DamselRescuingKnight` останется непротестированным.

Связь классов – это «двуглавый зверь». С одной стороны, сильно связанный код сложен в тестировании, его трудно использовать повторно, он сложен в понимании, и, как правило, такой код оказывается очень хрупким при исправлении ошибок (исправление одной ошибки может привести к нескольким новым). С другой стороны, совершенно не связанный код ничего не делает. Чтобы делать что-то полезное, классы должны знать друг о друге. Связанность необходима, но должна тщательно контролироваться.

С другой стороны, благодаря DI объекты получают свои зависимости во время создания от некоторой третьей стороны, координирующей работу каждого объекта в системе. Объекты не создают и не получают свои зависимости самостоятельно – зависимости внедряются в объекты.

Для иллюстрации рассмотрим класс `BraveKnight`, представленный в листинге 1.4, реализующий рыцаря, который не только храбр, но и способен совершать любые подвиги.

Листинг 1.4. Класс `BraveKnight`, достаточно гибкий, чтобы совершить любой подвиг

```
package com.springinaction.knights;

public class BraveKnight implements Knight {
```

¹ Здесь необходимо уточнить, что класс `DamselRescuingKnight` реализует «рыцаря, спасающего даму», а класс `RescueDamselQuest` реализует «сценарий спасения дамы». – Прим. перев.

```
private Quest quest;

public BraveKnight(Quest quest) {
    this.quest = quest; // Внедрение сценария подвига
}

public void embarkOnQuest() throws QuestException {
    quest.embark();
}
}
```

Как видно из листинга, в отличие от класса `DamselRescuingKnight`, класс `BraveKnight` не создает собственного сценария подвига, а получает его извне, в виде аргумента конструктора. Такой способ внедрения зависимостей называется *внедрением через конструктор*.

Более того, сценарий подвига имеет тип интерфейса `Quest`, который реализуют все такие сценарии. Поэтому `BraveKnight` (храбрый рыцарь) сможет совершать любые подвиги, такие как `RescueDamselQuest` (спасти даму), `SlayDragonQuest` (убить дракона), `MakeRoundTableRounderQuest` (стать рыцарем Круглого стола) или любой другой, реализующий интерфейс `Quest`.

Фактически класс `BraveKnight` никак не связан с конкретной реализацией `Quest`. Для него не важно, какой подвиг будет поручен, при условии что он реализует интерфейс `Quest`. В этом состоит основное преимущество DI – слабая связанность. Если объект взаимодействует со своими зависимостями только через их интерфейсы (ничего не зная о конкретных реализациях или особенностях их создания), зависимости можно будет замещать любыми другими реализациями, без необходимости учитывать эти различия в самом объекте.

Прием замены зависимостей очень часто используется при тестировании, когда выполняется подстановка фиктивной реализации. Класс `DamselRescuingKnight` невозможно было протестировать в полной мере из-за тесной связи, но класс `BraveKnight` легко поддается тестированию за счет подстановки фиктивной реализации интерфейса `Quest`, как показано в листинге 1.5.

Листинг 1.5. Протестировать класс `BraveKnight` можно с помощью фиктивной реализации интерфейса `Quest`

```
package com.springinaction.knights;

import static org.mockito.Mockito.*;
```

```

import org.junit.Test;

public class BraveKnightTest {
    @Test
    public void knightShouldEmbarkOnQuest() throws QuestException {
        Quest mockQuest = mock(Quest.class); // Создание фиктивного
                                              // объекта Quest
        BraveKnight knight = new BraveKnight(mockQuest); // Внедрение
        knight.embarkOnQuest();

        verify(mockQuest, times(1)).embark();
    }
}

```

В данном примере фиктивная реализация интерфейса `Quest` создана с помощью фреймворка Mockito. После получения фиктивного объекта создается новый экземпляр `BraveKnight`, в который через конструктор внедряется фиктивный объект `Quest`. После вызова метода `embarkOnQuest()` выполняется обращение к фреймворку Mockito с целью убедиться, что метод `embark()` интерфейса `Quest` фиктивного объекта был вызван только один раз.

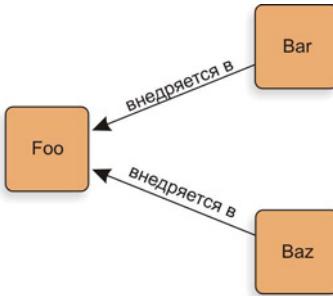


Рис. 1.1. Механизм внедрения зависимостей основан на предоставлении объекту его зависимостей извне, а не на приобретении этих зависимостей самим объектом

Передача сценария подвига рыцарю

Теперь, когда класс `BraveKnight` может принимать любые задания, как определить, какой именно объект `Quest` был ему передан? Процесс создания связей между прикладными компонентами называется *связыванием* (wiring). Фреймворк Spring поддерживает множест-

во способов связывания компонентов, но наиболее общим из них является способ на основе XML. В листинге 1.6 показано содержимое простого конфигурационного файла Spring, `knights.xml`, который передает объекту `BraveKnight` задание `SlayDragonQuest`.

Листинг 1.6. Внедрение сценария `SlayDragonQuest` в объект `BraveKnight` средствами Spring

```
<?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-3.0.xsd">

    <bean id="knight" class="com.springinaction.knights.BraveKnight">
        <constructor-arg ref="quest" />    <!-- Внедрение компонента quest --&gt;
    &lt;/bean&gt;
    &lt;!-- Создание SlayDragonQuest --&gt;
    &lt;bean id="quest"
          class="com.springinaction.knights.SlayDragonQuest" /&gt;
&lt;/beans&gt;</pre>
```

Это простой способ связывания компонентов. Пока не стоит слишком беспокоиться о деталях. Подробнее о конфигурировании Spring и о том, что происходит на данном этапе, будет рассказываться в главе 2, где также будут показаны другие способы связывания компонентов в Spring.

Теперь, объявив отношения между `BraveKnight` и `Quest`, необходимо загрузить XML-файл и запустить приложение.

Рассмотрим этот механизм в действии

В приложении, созданном на основе Spring, *контекст приложения* загружает определения компонентов и связывает их вместе. За создание объектов, составляющих приложение, и их связывание полностью отвечает контекст приложения. В составе фреймворка Spring имеется несколько реализаций контекста приложения.

Поскольку компоненты приложения объявлены в XML-файле `knights.xml`, в качестве контекста приложения может использоваться класс `ClassPathXmlApplicationContext`. Реализация контекста в Spring загружает контекст из одного или более XML-файлов, находящихся в библиотеке классов (`classpath`). Метод `main()` в листинге 1.7 использует `ClassPathXmlApplicationContext`, чтобы загрузить `knight.xml` и получить ссылку на объект `Knight`.



Листинг 1.7. KnightMain.java загружает контекст Spring, содержащий объект Knight

```
package com.springinaction.knights;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class KnightMain {
    public static void main(String[] args) {
        // Загрузка контекста Spring
        ApplicationContext context =
            new ClassPathXmlApplicationContext("knights.xml");

        // Получение компонента knight
        Knight knight = (Knight) context.getBean("knight");
        // Использование компонента knight
        knight.embarcOnQuest();
    }
}
```

Здесь метод `main()` создает контекст приложения Spring на основе файла `knights.xml`. Затем использует контекст как фабрику для извлечения компонента с идентификатором «`knight`». Получив ссылку на объект `Knight`, он вызывает метод `embarkOnQuest()`, чтобы отправить рыцаря выполнять задание. Обратите внимание, что этот класс ничего не знает о задании `Quest`, переданном рыцарю. Собственно говоря, он даже не знает, что имеет дело с классом `BraveKnight`. Только файл `knights.xml` имеет полную информацию о реализациях, участвующих в работе.

На этом мы закончим краткое знакомство с приемом внедрения зависимостей. Дополнительные примеры применения DI будут встречаться на протяжении всей книги. Желающим поближе познакомиться с этим приемом я рекомендую прочитать книгу Дханжи Прасанна (*Dhanji R. Prasanna*) «Dependency Injection».

А теперь познакомимся с другими стратегиями упрощения программирования на языке Java – декларативным программированием посредством *аспектов*.

1.1.3. Применение аспектно-ориентированного программирования

Хотя DI делает возможным ослабить связь между компонентами приложения, *аспектно-ориентированное программирование* позво-

ляет оформлять функциональность, используемую в приложении, в виде многократно используемых компонентов.

Аспектно-ориентированное программирование часто определяют как прием программирования, поддерживающий разделение задач в пределах программной системы. Системы конструируются из нескольких компонентов, каждый из которых отвечает за определенную часть функциональности. Зачастую эти компоненты также несут дополнительную ответственность сверх своей основной функциональности. Системные службы, такие как журналирование, управление транзакциями и безопасность, часто находят свое отражение в компонентах, основная задача которых заключается в чем-то другом. Такие системные службы обычно называют *сквозными задачами*, потому что в их работу может вовлекаться несколько компонентов системы.

Распространение этих задач на несколько компонентов влечет за собой увеличение сложности программного кода:

- программный код, реализующий решение общесистемных задач, дублируется в разных компонентах. Это означает, что в случае необходимости что-то изменить в этой реализации придется просмотреть множество компонентов. Даже если вынести решение задачи в отдельный модуль, чтобы компонентам оставалось только вызывать единственный метод, все равно вызов этого метода будет дублироваться в разных местах;
- компоненты захламляются программным кодом, не имеющим прямого отношения к их основной функциональности. Метод добавления записи в адресную книгу должен заботиться только о том, как добавить адрес, а не о безопасности или о необходимости использования транзакции.

Эта сложность иллюстрируется на рис. 1.2. Бизнес-объекты слева слишком тесно связаны с системными службами. Мало того, что каждый объект знает, что его операции должны фиксироваться в журнале, соответствовать требованиям безопасности и выполнятьсь в рамках транзакции, они еще несут ответственность за использование этих служб.

АОР делает возможным отделение этих служб и декларативное их применение к необходимым компонентам. Благодаря этому компоненты могут сосредоточиться на решении собственных задач, полностью игнорируя системные службы, которые могут быть вовлечены в общий процесс. Помимо говоря, аспекты позволяют сохранить простоту POJO.

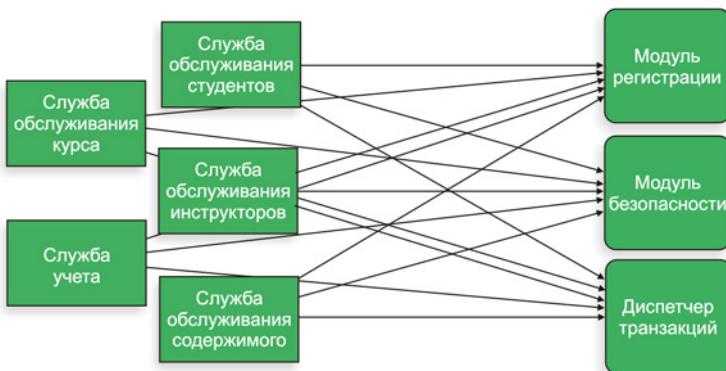


Рис. 1.2. Вызовы общесистемных задач, таких как журналирование и безопасность, часто разбросаны по модулям, где эти задачи не являются основными

Аспекты можно представить как *обертки*, охватывающие множество компонентов в приложении, как показано на рис. 1.3. Ядро приложения составляют модули, реализующие основную логику его работы. Применяя AOP, можно обернуть ядро приложения дополнительными функциональными слоями. Эти слои могут применяться декларативно повсюду в приложении, при этом ядро приложения может даже не подозревать об их существовании. Эта очень мощная концепция

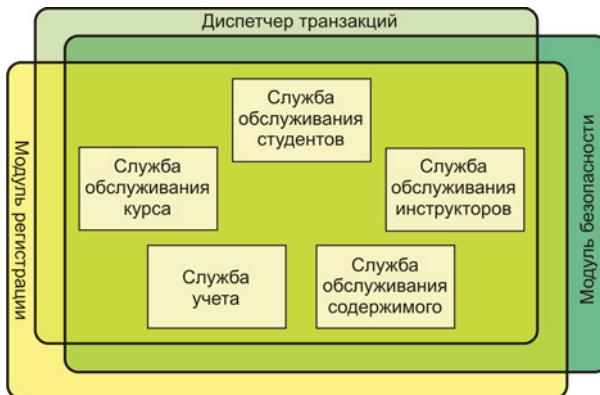


Рис. 1.3. Использование АОП, общесистемные задачи охватывают компоненты, на которые они воздействуют. Это позволяет прикладным компонентам фокусироваться на решении своих специфических задач

позволяет убрать из ядра приложения любые операции, связанные с безопасностью, управлением транзакциями или журналированием.

Для демонстрации особенностей применения аспектов в Spring вернемся к предыдущему примеру и добавим в него простой аспект.

AOP в действии

Сведения о рыцарях дошли до наших времен только потому, что их подвиги были отмечены в песнях тогдашних певцов и сочинителей баллад – менестрелей. Предположим, что вам потребовалось зафиксировать хронику приездов и отъездов вашего храброго рыцаря `BraveKnight` с помощью услуг такого менестреля. В листинге 1.8 представлен класс `Minstrel`, который можно было бы использовать для этой цели.

Листинг 1.8. Класс `Minstrel`, фиксирующий средневековую историю в песнях

```
package com.springinaction.knights;

public class Minstrel {
    public void singBeforeQuest() { // Вызывается перед выполнением задания
        System.out.println( Fa la la; The knight is so brave! );
    }

    public void singAfterQuest() { // Вызывается после выполнения задания
        System.out.println(
             Tee hee he; The brave knight did embark on a quest! );
    }
}
```

Как видно из листинга 1.8, класс `Minstrel` содержит всего два метода. Метод `singBeforeQuest()` вызывается перед отправкой рыцаря в поход, а метод `singAfterQuest()` – после его возвращения. Пока не видится никаких сложностей, мешающих работе этого кода, поэтому внесем соответствующие изменения в класс `BraveKnight`, чтобы он мог использовать компонент `Minstrel`. В листинге 1.9 представлена первая попытка.

Листинг 1.9. Класс `BraveKnight`, который должен вызывать методы класса `Minstrel`

```
package com.springinaction.knights;

public class BraveKnight implements Knight {
```



```
private Quest quest;
private Minstrel minstrel;

public BraveKnight(Quest quest, Minstrel minstrel) {
    this.quest = quest;
    this.minstrel = minstrel;
}

public void embarkOnQuest() throws QuestException {
    minstrel.singBeforeQuest(); // Должен ли рыцарь руководить
    quest.embark();           // своим менестрелем?
    minstrel.singAfterQuest();
}
```

Это должно сработать. Но что-то тут не так. Действительно ли рыцарь должен беспокоиться о руководстве своим менестрелем? Мне кажется, что менестрель должен просто делать свою работу, без напоминаний со стороны рыцаря. В конце концов, менестрель сам должен позаботиться о том, чтобы воспеть подвиги своего рыцаря. С какой стати рыцарь должен напоминать менестрелю, чтобы тот не забывал выполнять свою работу?

Кроме того, из-за того что рыцарь должен знать о существовании менестреля, мы вынуждены внедрять компонент `Minstrel` в компонент `BraveKnight`. Это не только усложняет реализацию класса `BraveKnight`, но также заставляет задаться вопросом: может ли рыцарь обойтись без менестреля? Что, если ссылка `minstrel` будет иметь значение `null`? Следует ли предусмотреть такой случай и проверять значение ссылки?

Простой класс `BraveKnight` начинает становиться все более сложным и мог бы стать еще сложнее, если бы в него пришлось добавить проверку ссылки `minstrel` на равенство значению `null`. Но благодаря использованию АОР можно просто объявить, что менестрель должен воспевать подвиги рыцаря и освободить рыцаря от необходимости руководить действиями менестреля.

Чтобы превратить класс `Minstrel` в аспект, достаточно просто объявить его таковым в конфигурационном файле Spring. В листинге 1.10 представлена измененная версия файла `knights.xml`, объявляющая класс `Minstrel` аспектом.

Листинг 1.10. Объявление класса Minstrel аспектом

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
    <bean id="knight" class="com.springinaction.knights.BraveKnight">
        <constructor-arg ref="quest" />
    </bean>

    <bean id="quest"
        class="com.springinaction.knights.SlayDragonQuest" />

    <!-- Объявление компонента Minstrel -->
    <bean id="minstrel"
        class="com.springinaction.knights.Minstrel" />

<aop:config>
    <aop:aspect ref="minstrel">
        <!-- Объявление точки внедрения -->
        <aop:pointcut id="embark"
            expression="execution(* *.embarkOnQuest(..))" />

        <aop:before pointcut-ref="embark"
            method="singBeforeQuest"/> <!-- Операция, выполняемая до -->

        <aop:after pointcut-ref="embark"
            method="singAfterQuest"/> <!-- Операция, выполняемая после -->
    </aop:aspect>
</aop:config>
</beans>
```

Здесь, для объявления компонента Minstrel аспектом, используется пространство имен aop конфигураций Spring. Во-первых, объект Minstrel необходимо объявить компонентом. Затем сослаться на него в элементе `<aop:aspect>`. Определяя аспект далее, необходимо объявить (с помощью `<aop:before>`), что перед вызовом метода `embarkOnQuest()` должен вызываться метод `singBeforeQuest()` компо-

нента Minstrel. Это называется *объявлением предварительной операции* (before advice). А затем объявить (с помощью `<aop:after>`), что после вызова метода `embarkOnQuest()` должен вызываться метод `singAfterQuest()`. Это называется *объявлением последующей операции* (after advice).

В обоих случаях атрибут `pointcut-ref` должен ссылаться на *точку внедрения* с именем `embark`. Эта точка внедрения определяется предшествующим элементом `<pointcut>`, атрибуту `expression` которого присваивается выражение, определяющее точки внедрения операций. Синтаксис выражения определяется синтаксисом языка выражений AspectJ точек внедрения.

Не волнуйтесь, если вы незнакомы с языком AspectJ или с особенностями записи выражений определения точек внедрения. Подробнее тема аспектно-ориентированного программирования в Spring будет рассматриваться в главе 5. А пока достаточно знать, что этими объявлениями фреймворку Spring предлагается вызывать методы `singBeforeQuest()` и `singAfterQuest()` компонента `Minstrel` до и после вызова метода `embarkOnQuest()` компонента `BraveKnight`.

Вот и все! Добавив немного кода разметки XML, мы только что превратили компонент `Minstrel` в аспект Spring. Не беспокойтесь, если здесь что-то осталось непонятным – в главе 5 еще будет приводиться множество примеров аспектно-ориентированного программирования в Spring. А сейчас рассмотрим два важных следствия из этого примера.

Во-первых, компонент `Minstrel` так и остался простым Java-объектом (POJO) – ничто в нем не указывает, что он должен использоваться как аспект. Вместо этого компонент `Minstrel` был превращен в аспект декларативно в контексте Spring.

Во-вторых, что особенно важно, компонент `Minstrel` можно применить к компоненту `BraveKnight` вообще без его ведома. Фактически `BraveKnight` вообще не подозревает о существовании компонента `Minstrel`.

Следует также отметить, что прежде чем использовать волшебство Spring по превращению объекта `Minstrel` в аспект, его сначала необходимо объявить компонентом Spring с помощью элемента `<bean>`. Дело в том, что с аспектами Spring можно выполнять любые операции, которые можно выполнять с другими компонентами Spring, такие как внедрение их в виде зависимостей.

С помощью аспектов воспевать подвиги рыцаря стало интереснее. Но поддержку аспектно-ориентированного программирования

в Spring можно использовать для решения более практических задач. Как будет показано ниже, Spring AOP можно использовать для реализации таких служб, как декларативные транзакции (глава 7) и безопасность (глава 10).

А пока познакомимся с еще одним приемом Spring, упрощающим разработку на языке Java.

1.1.4. Устранение шаблонного кода с помощью шаблонов

Приходилось ли вам когда-нибудь писать некоторый программный код и затем испытывать ощущение, что вы уже писали его прежде? Это не дежавю, друзья мои. Это – шаблонный код, то есть код, который часто приходится писать снова и снова, чтобы реализовать типичную и простую задачу.

К сожалению, в Java API существует масса мест, где приходится писать огромное количество шаблонного кода. Типичный пример шаблонного кода можно увидеть в приложениях, использующих JDBC для работы с базами данных. Например, если вам доводилось работать с JDBC прежде, вы наверняка писали нечто подобное, представленное в листинге 1.11.

Листинг 1.11. Многие Java API, такие как JDBC, требуют писать массу шаблонного кода

```
public Employee getEmployeeById(long id) {
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(
            "select id, firstname, lastname, salary from " +
            "employee where id=?");           // Выбрать сотрудника
        stmt.setLong(1, id);
        rs = stmt.executeQuery();
        Employee employee = null;
        if (rs.next()) {
            employee = new Employee();    // Создать объект из данных
            employee.setId(rs.getLong("id"));
            employee.setFirstName(rs.getString("firstname"));
            employee.setLastName(rs.getString("lastname"));
            employee.setSalary(rs.getBigDecimal("salary"));
        }
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        if (stmt != null)
            try { stmt.close(); } catch (SQLException e) {}
        if (conn != null)
            try { conn.close(); } catch (SQLException e) {}
        if (rs != null)
            try { rs.close(); } catch (SQLException e) {}
    }
}
```

```
        }
        return employee;
    } catch (SQLException e) { // Что здесь следует сделать?
        }
    } finally {
        if(rs != null) { // Освободить ресурсы
            try {
                rs.close();
            } catch(SQLException e) {}
        }

        if(stmt != null) {
            try {
                stmt.close();
            } catch(SQLException e) {}
        }

        if(conn != null) {
            try {
                conn.close();
            } catch(SQLException e) {}
        }
    }
    return null;
}
```

Программный код в листинге 1.11 извлекает имя работника и сумму его зарплаты из базы данных с использованием JDBC. Но, держу пари, вам пришлось напрячься, чтобы увидеть это, потому что реализация фактической логики выполнения запроса похоронена под массой программного кода, реализующего обряд взаимодействия с JDBC. Сначала нужно создать соединение с базой данных, затем определить запрос и, наконец, выполнить запрос на получение результатов. А чтобы умиротворить гнев JDBC, требуется обеспечить обработку исключения `SQLException`, даже при том, что практически ничего не требуется делать в случае его появления.

Наконец, после всего сказанного и сделанного необходимо еще освободить ресурсы, закрыть соединение, освободить объект запроса и набор результатов. Причем все это также может вызвать гнев JDBC. То есть и при выполнении этих операций необходимо предусмотреть обработку исключения `SQLException`.

Наиболее заметной особенностью листинга 1.11 является большое количество программного кода, который практически без изменений

придется писать для реализации любых операций с JDBC. Лишь малая его часть имеет прямое отношение к запросу информации, основная же часть – шаблонный код обслуживания требований JDBC.

JDBC – не единственный источник необходимости писать шаблонный код. Многие API требуют писать аналогичный шаблонный код. Операции с JMS, JNDI и со службами REST часто требуют писать массу повторяющегося кода.

Фреймворк Spring стремится помочь в устраниении шаблонного кода путем заключения его в шаблоны. Класс `JdbcTemplate` из фреймворка Spring позволяет выполнять операции с базой данных без лишних церемоний, требуемых традиционным JDBC.

Например, с помощью класса `SimpleJdbcTemplate` (специализированный наследник класса `JdbcTemplate`, использующий преимущества особенностей Java 5) метод `getEmployeeById()` можно переписать так, что он окажется сосредоточен исключительно на задаче извлечения данных о работнике, а не на удовлетворении требований JDBC API. В листинге 1.12 показано, как может выглядеть обновленная версия метода `getEmployeeById()`.

Листинг 1.12. Шаблоны позволяют сосредоточить все внимание на решении предметных задач

```
public Employee getEmployeeById(long id) {  
    return jdbcTemplate.queryForObject(  
        "select id, firstname, lastname, salary " + // SQL-запрос  
        "from employee where id=?",
        new RowMapper<Employee>() {
            public Employee mapRow(ResultSet rs,
                int rowNum) throws SQLException { // Отображение
                Employee employee = new Employee(); // результатов в объект
                employee.setId(rs.getLong("id"));
                employee.setFirstName(rs.getString("firstname"));
                employee.setLastName(rs.getString("lastname"));
                employee.setSalary(rs.getBigDecimal("salary"));
                return employee;
            }
        },
        id); // Параметры запроса
}
```

Как видите, новая версия `getEmployeeById()` намного проще и сосредоточена только на выборе информации из базы данных. Метод `queryForObject()` шаблона принимает строку SQL-запроса, объект

RowMapper (отображающий набор полученных данных в объект предметной области) и ноль или более параметров запроса. Чего нет в методе `getEmployeeById()`, так это шаблонного кода обслуживания JDBC. Все необходимые действия выполняются шаблоном.

Я показал, как фреймворк Spring борется со сложностью разработки приложений на языке Java, предоставляя возможность использовать POJO, внедрение зависимостей, AOP и шаблоны. Попутно было показано, как конфигурировать компоненты и аспекты в XML-файлах. Но как загружаются эти файлы? И куда они загружаются? Познакомимся с контейнером Spring – местом, где располагаются компоненты приложения.

1.2. Контейнер компонентов

В приложениях на основе фреймворка Spring прикладные объекты располагаются внутри контейнера Spring. Как показано на рис. 1.4, контейнер создает объекты, связывает их друг с другом, конфигурирует и управляет их полным жизненным циклом, от зарождения до самой их смерти (или от оператора `new` до вызова метода `finalize()`).

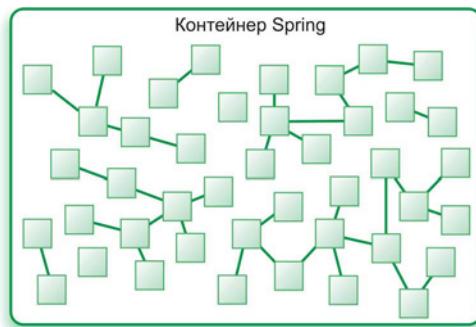


Рис. 1.4. Объекты в приложениях на основе фреймворка Spring создаются, связываются между собой и существуют внутри контейнера Spring

В следующей главе будет показано, как настроить фреймворк Spring, чтобы он знал, какие объекты следует создать, как их конфигурировать и связывать между собой. Но сначала необходимо дать знать контейнеру, где будут располагаться объекты. Понимание

особенностей функционирования контейнера поможет понять, как производится управление объектами.

Контейнер находится в ядре фреймворка Spring Framework. Для управления компонентами, составляющими приложение, он использует прием внедрения зависимостей (DI). Управление включает создание взаимосвязей между взаимодействующими компонентами. Фактически эти объекты яснее и проще для понимания, поддерживают возможность повторного использования и легко поддаются тестированию.

Фреймворк Spring имеет не один контейнер. В его состав входят несколько реализаций контейнера, которые подразделяются на два разных типа. *Фабрики компонентов* (bean factories) (определяются интерфейсом `org.springframework.beans.factory.BeanFactory`) – самые простые из контейнеров, обеспечивающие базовую поддержку DI. *Контекст приложений* (application contexts) (определяется интерфейсом `org.springframework.context.ApplicationContext`) основан на понятии фабрик компонентов и реализует прикладные службы фреймворка, такие как возможность приема текстовых сообщений из файлов свойств и возможность подписывать другие программные компоненты на события, возникающие в приложении.

С фреймворком Spring можно работать, используя и фабрики компонентов, и контексты приложений, но для большинства приложений фабрики компонентов часто оказываются слишком низкоуровневым инструментом. Поэтому контексты приложений выглядят более предпочтительно, чем фабрики компонентов. Далее мы сосредоточимся на использовании контекстов приложения и не будем тратить время на обсуждение фабрик компонентов.

1.2.1. Работа с контекстом приложения

В составе Spring имеется несколько разновидностей *контекстов приложений*. Три из них используются наиболее часто:

- ❑ `ClassPathXmlApplicationContext` – загружает определение контекста из XML-файла, расположенного в библиотеке классов (classpath), и обрабатывает файлы с определениями контекстов как ресурсы;
- ❑ `FileSystemXmlApplicationContext` – загружает определение контекста из XML-файла в файловой системе;
- ❑ `XmlWebApplicationContext` – загружает определение контекста из XML-файла, содержащегося внутри веб-приложения.

Подробнее о XmlWebApplicationContext будет рассказываться в главе 8 вместе с обсуждением веб-приложений на основе фреймворка Spring. А пока просто загрузим контекст приложения из файловой системы, используя FileSystemXmlApplicationContext, или из библиотеки классов (classpath), используя ClassPathXmlApplicationContext.

Загрузка контекста приложения из файловой системы или из библиотеки классов похожа на загрузку компонентов с использованием фабрики компонентов. Например, ниже показано, как можно использовать FileSystemXmlApplicationContext:

```
ApplicationContext context = new
    FileSystemXmlApplicationContext("c:/foo.xml");
```

Аналогично выполняется загрузка контекста приложения из библиотеки классов приложения с помощью ClassPathXmlApplicationContext:

```
ApplicationContext context = new
    ClassPathXmlApplicationContext("foo.xml");
```

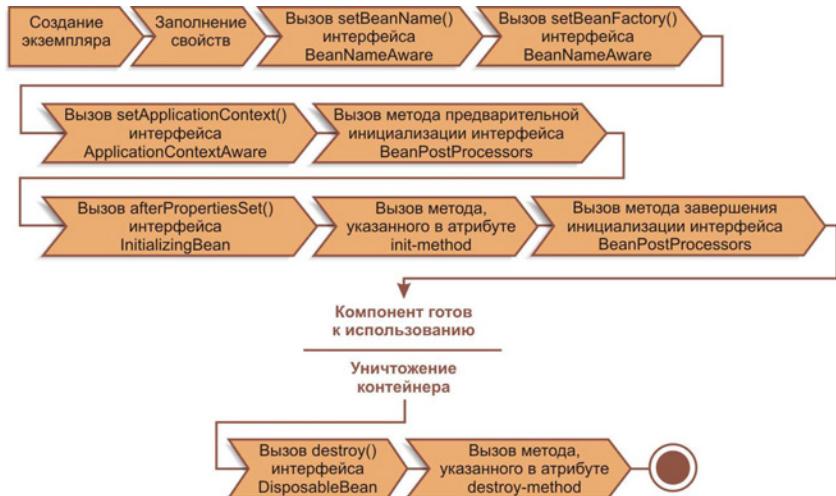


Рис. 1.5. От создания до уничтожения в контейнере Spring компонент преодолевает несколько этапов. Spring позволяет настроить выполнение каждого из этапов

Различие между `FileSystemXmlApplicationContext` и `ClassPathXmlApplicationContext` состоит в том, что `FileSystemXmlApplicationContext` будет искать файл `foo.xml` в определенном месте, внутри файловой системы, тогда как `ClassPathXmlApplicationContext` будет искать `foo.xml` по всей библиотеке классов (включая JAR-файлы).

После загрузки контекста приложения извлекать компоненты из контейнера Spring можно с помощью метода `getBean()` контекста.

Теперь, после знакомства с основами создания контейнера Spring, познакомимся поближе с жизненным циклом компонента в контейнере.

1.2.2. Жизненный цикл компонента

В традиционных Java-приложениях жизненный цикл компонента довольно прост. Сначала компонент создается с помощью ключевого слова `new` (при этом, возможно, выполняется его десериализация), после чего он готов к использованию. Когда компонент перестает использоваться, он утилизируется сборщиком мусора и в конечном счете попадает в большой «битоприемник» на небесах.

Напротив, жизненный цикл компонента внутри контейнера Spring намного сложнее. Иметь представление о жизненном цикле компонента в контейнере Spring очень важно, потому что в этом случае появляются новые возможности управления процессом создания компонента. Рисунок 1.5 иллюстрирует начальные этапы жизненного цикла типичного компонента, которые он минует, прежде чем будет готов к использованию.

Как показано на рисунке, фабрика компонентов выполняет несколько подготовительных операций, перед тем как компонент будет готов к использованию. Исследуем рис. 1.5 более детально.

1. Spring создает экземпляр компонента.
2. Spring внедряет значения и ссылки на компоненты в свойства данного компонента.
3. Если компонент реализует интерфейс `BeanNameAware`, Spring передает идентификатор компонента методу `setBeanName()`.
4. Если компонент реализует интерфейс `BeanFactoryAware`, Spring вызывает метод `setBeanFactory()`, передавая ему саму фабрику компонентов.
5. Если компонент реализует интерфейс `ApplicationContextAware`, Spring вызывает метод `setApplicationContext()`, передавая ему ссылку на вмещающий контекст приложения.



6. Если какие-либо из компонентов реализуют интерфейс BeanPostProcessor, Spring вызывает их методы postProcessBeforeInitialization().
7. Если какие-либо из компонентов реализуют интерфейс InitializingBean, Spring вызывает их методы afterPropertiesSet(). Аналогично, если компонент был объявлен с атрибутом init-method, вызывается указанный метод инициализации.
8. Если какие-либо из компонентов реализуют интерфейс BeanPostProcessor, Spring вызывает их методы postProcessAfterInitialization().
9. В этот момент компонент готов к использованию приложением и будет сохраняться в контексте приложения, пока он не будет уничтожен.
10. Если какие-либо из компонентов реализуют интерфейс DisposableBean, Spring вызывает их методы destroy(). Аналогично, если компонент был объявлен с атрибутом destroy-method, вызывается указанный метод.

Теперь вы знаете, как создавать и загружать *контейнер* Spring. Но пустой контейнер сам по себе не имеет практической пользы – он останется пустым, если в него ничего не поместить. Чтобы воспользоваться преимуществами Spring DI, необходимо связать между собой прикладные объекты в контейнере Spring. Подробнее о связывании компонентов рассказывается в главе 2.

Но сначала познакомимся с современным ландшафтом Spring и посмотрим, из чего состоит фреймворк Spring Framework и что могут предложить последние его версии.

1.3. Обзор возможностей Spring

Как вы уже знаете, основной целью фреймворка Spring Framework является упрощение разработки корпоративных приложений на языке Java за счет использования приемов внедрения зависимостей, аспектно-ориентированного программирования и устранения необходимости писать шаблонный программный код. Даже если бы возможности фреймворка Spring ограничивались этим набором, его уже стоило бы использовать. Но в Spring заложены гораздо более широкие возможности, чем видно на поверхности.

При работе с фреймворком Spring Framework можно обнаружить еще несколько скрытых путей, упрощающих разработку на языке Java. Однако за пределами Spring Framework существует гигантская

экосистема проектов, построенных на базе основного фреймворка и расширяющих возможность применения Spring на такие области, как веб-службы, OSGi, Flash и даже .NET.

Для начала попробуем разложить основной фреймворк Spring Framework на составные части, чтобы посмотреть, что имеется в нашем распоряжении. А затем расширим наш кругозор и познакомимся с другими элементами в пухлом портфеле Spring.

1.3.1. Модули Spring

Фреймворк Spring состоит из нескольких *модулей*. После загрузки и распаковки архива с дистрибутивом Spring Framework можно обнаружить 20 различных JAR-файлов, как показано на рис. 1.6.

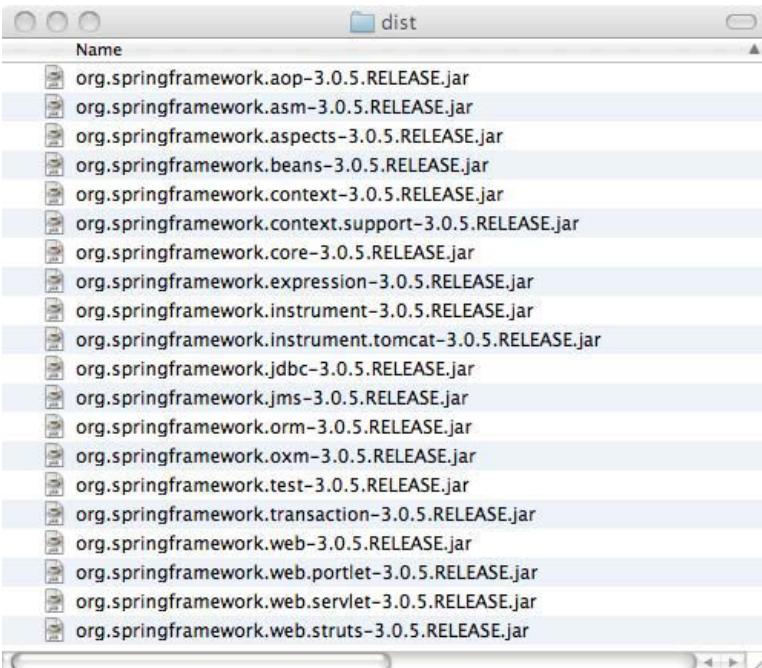


Рис. 1.6. 20 JAR-файлов, входящих в состав дистрибутива Spring Framework

20 JAR-файлов, составляющих Spring, распределяются по шести функциональным категориям, как показано на рис. 1.7.

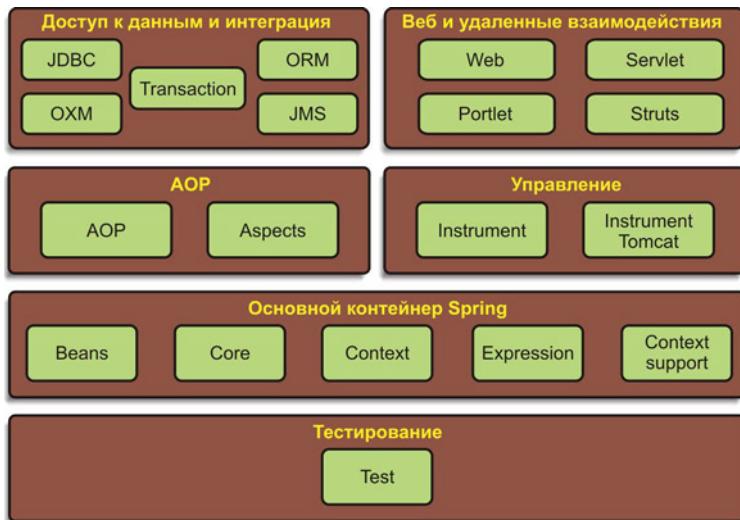


Рис. 1.7. Фреймворк Spring Framework состоит из модулей, подразделяющихся на шесть категорий

В совокупности эти модули-категории предоставляют все необходимое для разработки корпоративных приложений. Но никто не обязывает основывать приложение исключительно на фреймворке Spring. Вы свободны в выборе модулей, необходимых в приложении, и в поиске альтернатив, если Spring не отвечает всем требованиям. Фактически фреймворк Spring имеет точки интеграции с некоторыми другими фреймворками и библиотеками, что устраниет необходимость писать их самостоятельно.

Рассмотрим все модули Spring по отдельности, чтобы увидеть, как они вписываются в общую архитектуру фреймворка.

Основной контейнер Spring

Центральное положение в фреймворке Spring занимает *контейнер*, управляющий процессом создания и настройки компонентов приложения. Этот модуль содержит фабрику компонентов, обеспечивающую внедрение зависимостей. На фабрике компонентов появляются несколько реализаций контекста приложения Spring, каждый из которых предоставляет различные способы конфигурирования Spring.

В дополнение к *фабрике компонентов* и *контексту приложения* этот модуль также предоставляет несколько корпоративных служб,

таких как электронная почта, доступ к JNDI, интеграция с EJB и выполнение заданий по расписанию.

Как показано на рис. 1.7, все модули Spring основаны на реализации основного контейнера. Эти классы неявно используются на этапе настройки приложения. Модули основного контейнера будут обсуждаться на протяжении всей книги, начиная с главы 2, где детально будет рассматриваться прием внедрения зависимостей.

Модуль AOP

Фреймворк Spring обеспечивает богатую поддержку аспектно-ориентированного программирования в своем модуле *AOP*. Данный модуль служит основой при разработке аспектов в приложении, построенном на основе Spring. Как и DI, AOP способствует ослаблению связей между прикладными объектами. Однако поддержка AOP позволяет отделять аспекты приложения (такие как транзакции и безопасность) от объектов, к которым они применяются. Более подробно поддержка AOP будет рассматриваться в главе 5.

Доступ к данным и интеграция

Работа с JDBC зачастую сводится к обширному использованию шаблонного кода, который устанавливает соединение, создает SQL-запрос, обрабатывает результаты запроса, а затем закрывает соединение. Модуль поддержки JDBC *объектов доступа к данным* (Data Access Objects, DAO) в Spring абстрагирует шаблонный код и позволяет сохранить простым и прозрачным программный код, реализующий операции с базами данных, а также предотвратить проблемы, возникающие в результате ошибки освобождения ресурсов. Этот модуль также образует слой важных исключений, основанных на сообщениях об ошибках, посылаемых некоторыми серверами баз данных. Благодаря этому вам больше не придется расшифровывать непонятные сообщения об ошибках SQL!

Для тех, кто предпочитает использовать инструменты *объектно-реляционного отображения* (Object-Relational Mapping, ORM) поверх JDBC, фреймворк Spring предоставляет модуль ORM. Поддержка ORM в Spring основана на поддержке DAO, обеспечивающей удобный способ создания объектов доступа к данным для некоторых ORM-решений. Фреймворк Spring не пытается реализовать свое собственное ORM-решение, а просто предоставляет рычаги управления некоторыми популярными фреймворками ORM, включая Hibernate, Java Persistence API, Java Data Objects и iBATIS SQL



Maps. Механизм управления транзакциями в Spring поддерживает все эти фреймворки ORM наряду с JDBC.

В главе 6 будет показано, как абстракция операций с JDBC в Spring может существенно упростить программный код, реализующий доступ к данным.

Этот модуль также включает абстракцию интерфейса Java доступа к службам обмена сообщениями (Java Message Service, JMS) для обеспечения асинхронных взаимодействий с другими приложениями посредством обмена сообщениями. Начиная с версии Spring 3.0, этот модуль еще включает механизм отображения объектов в формат XML, который прежде являлся частью проекта Spring Web Services.

Кроме того, данный модуль использует модуль АОР для предоставления службы управления транзакциями в приложении на основе фреймворка Spring. Поддержка транзакций подробно будет рассматриваться в главе 6.

Веб и удаленные взаимодействия

Параидигма *модель–представление–контроллер* (Model–View–Controller, MVC) часто используется при создании веб-приложений, в которых пользовательский интерфейс отделен от логики работы приложения. Язык Java не испытывает недостатка в фреймворках MVC, благодаря поддержке Apache Struts, JSF, WebWork и Tapestry, являющихся наиболее популярными реализациями MVC.

Несмотря на то что фреймворк Spring легко интегрируется с различными популярными фреймворками MVC, его модуль поддержки веб- и удаленных взаимодействий включает собственный фреймворк MVC, который использует приемы создания слабо связанных объектов в веб-слое приложения. Этот фреймворк имеет две разновидности: фреймворк на основе сервлетов, для создания обычных веб-приложений, и фреймворк на основе портлетов, для создания приложений на основе API Java-портлетов.

В дополнение к поддержке создания пользовательского интерфейса в веб-приложениях этот модуль также предоставляет поддержку удаленных взаимодействий для создания приложений, взаимодействующих с другими приложениями. В состав средств удаленных взаимодействий в Spring входят механизм *вызыва удаленных методов* (Remote Method Invocation, RMI), Hessian, Burlap, JAX-WS и собственный механизм вызова через протокол HTTP.

Подробнее фреймворк Spring MVC будет рассматриваться в главе 7. А в главе 10 мы познакомимся с организацией удаленных взаимодействий.

Тестирование

Учитывая важность тестов, написанных разработчиками, в состав фреймворка Spring был включен модуль поддержки тестирования приложений на основе Spring.

Внутри этого модуля можно обнаружить *коллекцию фиктивных объектов* для применения в модульных тестах, проверяющих работу с JNDI, сервлетами и портлетами. Для нужд интеграционного тестирования этот модуль предлагает поддержку загрузки коллекций компонентов в контекст приложения Spring и выполнение операций с этими компонентами.

Первый опыт использования модуля тестирования мы получим в главе 5. А затем, в главах 6 и 7, подкрепим полученные знания практикой тестирования операций доступа к данным и транзакций.

1.3.2. Дополнительные возможности Spring

При более детальном исследовании фреймворка Spring можно обнаружить намного больше, чем видно на поверхности. Фактически, помимо загружаемого дистрибутива Spring Framework, существует множество других проектов. Если остановиться только на использовании основного фреймворка Spring Framework, можно упустить богатейшие возможности, предлагаемые дополнительными проектами, расширяющими фреймворк Spring. Экосистема Spring включает несколько фреймворков и библиотек, построенных на основе базового фреймворка Spring Framework и друг на друге. Все вместе, вся экосистема Spring распространяет модель программирования Spring практически на все аспекты разработки на языке Java.

Потребовалось бы написать несколько томов, чтобы охватить всю экосистему Spring. Большая ее часть далеко выходит за рамки этой книги, тем не менее мы коротко познакомимся с некоторыми ее элементами. Ниже представлен краткий список того, что можно найти за пределами основного фреймворка Spring Framework.

Spring Web Flow

Фреймворк Spring Web Flow построен на основе фреймворка Spring MVC и обеспечивает поддержку создания диалоговых, многоэтапных веб-приложений, направляющих пользователя к его цели



(например, различные мастера или тележки с товарами в интернет-магазинах). Подробнее о фреймворке Spring Web Flow будет рассказываться в главе 9, а кроме того, дополнительную информацию можно получить на домашней странице проекта по адресу: <http://www.springsource.org/webflow>.

Spring Web Services

Несмотря на наличие поддержки декларативной публикации компонентов Spring в качестве веб-служб в основном фреймворке Spring Framework, с архитектурной точки зрения эти службы описываются на менее предпочтительную модель «contract-last»¹, когда определение службы создается на основе интерфейса компонента. Фреймворк Spring Web Services предлагает иную модель реализации веб-служб – «contract-first», когда программный код реализации пишется на основе определения службы.

В этой книге не будет рассказываться о фреймворке Spring-WS, но вы можете больше узнать о нем на домашней странице по адресу: <http://static.springsource.org/spring-ws/sites/2.0>.

Spring Security

Безопасность является важным аспектом многих приложений. Фреймворк Spring Security, реализованный на основе Spring AOP, предлагает декларативный механизм обеспечения безопасности приложений на основе фреймворка Spring. Как добавить поддержку Spring Security в приложения, будет показано в главе 10. Дальнейшие исследования этого фреймворка можно продолжить на домашней странице проекта, по адресу: <http://static.springsource.org/spring-security/site>.

Spring Integration

Многие корпоративные приложения должны взаимодействовать с другими корпоративными приложениями. Фреймворк Spring Integration предлагает реализацию нескольких распространенных шаблонов интеграции в декларативном стиле.

¹ Суть модели «contract-last» заключается в том, что при создании веб-службы сначала пишется программный код, реализующий ее, а затем на его основе создается WSDL-определение веб-службы. Существует противоположная ей модель, «contract-first», согласно которой сначала создается WSDL-определение службы, а затем на его основе пишется программный код реализации. – Прим. перев.

В этой книге не будет рассказываться о фреймворке Spring Integration. Но вы можете больше узнать о нем из книги «Spring Integration in Action» Марка Фишера (Mark Fisher), Джонаса Партнера (Jonas Partner), Мариуса Богоевича (Marius Bogoevici) и Ивейна Фулда (Iwein Fuld). Или посетив домашнюю страницу проекта по адресу: <http://www.springsource.org/spring-integration>.

Spring Batch

Когда требуется выполнить массив операций с данными, необходимо использовать инструмент пакетной обработки. Те, кто собирается заниматься разработкой приложения пакетной обработки данных, может воспользоваться фреймворком Spring Batch, дополняющим надежную модель Spring, ориентированную на создание простых Java-объектов.

Обсуждение фреймворка Spring Batch выходит далеко за рамки этой книги. Поэтому за дополнительной информацией о нем я отсылаю вас к книге Тьерри Темплиера (Thierry Templier) и Арно Коголагнеса (Arnaud Cogoluegnes) «Spring Batch in Action». Поближе познакомиться с Spring Batch можно на домашней странице проекта по адресу: <http://static.springsource.org/spring-batch>.

Spring Social

Социальные сети – нарастающая тенденция современного Интернета, и появляется все больше и больше приложений, поддерживающих интеграцию с сайтами социальных сетей, такими как Facebook и Twitter. Если эта тема заинтересует вас, подумайте об использовании фреймворка Spring Social, расширения Spring для поддержки социальных сетей.

Spring Social – относительно новый фреймворк и не рассматривается в этой книге, но вы можете больше узнать о нем на домашней странице по адресу: <http://www.springsource.org/spring-social>.

Spring Mobile

Мобильные приложения – еще одна обширная область разработки программного обеспечения. Все больше пользователей отдают предпочтение смартфонам и планшетным компьютерам. Фреймворк Spring Mobile – это новейшее расширение Spring для поддержки разработки мобильных веб-приложений.

С фреймворком Spring Mobile тесно связан проект Spring Android. Этот новый проект, созданный меньше месяца тому назад на момент



написания этих строк, ставит своей целью заимствовать простоту разработки на основе Spring Framework для создания приложений на платформе Android. Этот проект предлагает использовать Spring-версию класса RestTemplate (подробнее о классе RestTemplate рассказывается в главе 12), которую можно использовать в приложениях для Android.

Обсуждение данных проектов так же выходит за рамки книги «Spring в действии», но вы можете больше узнать о них по адресам: <http://www.springsource.org/spring-mobile> и <http://www.springsource.org/spring-android>.

Spring Dynamic Modules

Фреймворк Spring Dynamic Modules (Spring-DM) представляет собой сплав механизмов внедрения зависимостей Spring и динамических модулей OSGi. С помощью Spring-DM можно создавать приложения, состоящие из нескольких отдельных, тесно взаимодействующих, слабосвязанных модулей, которые декларативно создают и используют службы внутри фреймворка OSGi.

Следует отметить, что из-за огромного влияния технологии OSGi модель Spring-DM декларативного объявления служб OSGi была формализована в спецификации OSGi под названием «OSGi Blueprint Container». Кроме того, компания SpringSource выполнила перенос Spring-DM в проект Eclipse, в составе семейства Gemini проектов OSGi, и сейчас эта реализация известна под названием Gemini Blueprint.

Spring LDAP

В дополнение к внедрению зависимостей и AOP повсюду в фреймворке Spring Framework используется еще один прием – создание абстракций на основе шаблонов для излишне сложных операций, таких как запросы JDBC или обмен сообщениями посредством JMS. Фреймворк Spring LDAP обеспечивает аналогичный подход на основе шаблонов к использованию механизма LDAP, устранив необходимость использовать шаблонный код при выполнении LDAP-операций.

Более подробную информацию о Spring LDAP можно найти по адресу: <http://www.springsource.org/ldap>.

Spring Rich Client

Веб-приложения все чаще занимают нишу традиционных настольных приложений. Но если вы один из тех, кто занимается разработкой

приложений на основе библиотеки Swing, обратите внимание на фреймворк Spring Rich Client – богатый возможностями комплект инструментов, добавляющий мощь Spring в Swing.

Spring.NET

При переводе проекта на платформу .NET необязательно оставлять в стороне механизмы внедрения зависимостей и АОР. Фреймворк Spring.NET предлагает те же самые средства обеспечения слабой связанныности и аспектно-ориентированного программирования, что и Spring, но на платформе .NET.

В дополнение к базовым функциональным возможностям DI и AOP Spring.NET предлагает несколько модулей, упрощающих разработку приложений на платформе .NET, включая модули для работы с ADO.NET, NHibernate, ASP.NET и MSMQ.

Поближе познакомиться с фреймворком Spring.NET можно по адресу: <http://www.springframework.net>.

Spring-Flex

Технологии Flex и AIR компании Adobe являются одними из наиболее мощных в области разработки полнофункциональных интернет-приложений. Для организации взаимодействий этих пользовательских интерфейсов с программным кодом на языке Java на стороне сервера можно использовать технологию удаленных взаимодействий и обмена сообщениями, известную как BlazeDS. Пакет интеграции Spring-Flex позволяет приложениям, созданным на основе Flex и AIR, взаимодействовать с серверными компонентами Spring посредством BlazeDS. Он также включает расширение для поддержки Spring Roo, обеспечивающее быструю разработку Flex-приложений.

Свое знакомство с пакетом Spring-Flex можно начать со страницы <http://www.springsource.org/spring-flex>. Обратите также внимание на фреймворк Spring ActionScript: <http://www.springactionscript.org>, позволяющий использовать многие преимущества Spring Framework в ActionScript.

Spring Roo

Все больше и больше разработчиков начинают опираться в своей работе на фреймворк Spring, поэтому вокруг фреймворка Spring и родственных ему стало накапливаться множество идиом и приемов программирования. В то же время появились такие фреймворки, как



Ruby on Rails и Grails, использующие модель разработки на основе сценариев, упрощающую конструирование приложений.

Фреймворк Spring Roo реализует диалоговое окружение, обеспечивающее возможность быстрой разработки приложений на основе Spring, объединяя в себе наиболее удачные приемы, выработанные на протяжении нескольких последних лет.

Что отличает Spring Roo от всех остальных фреймворков быстрой разработки приложений, так это возможность писать программный код на языке Java, использующий Spring Framework. Это самое настоящее приложение на основе Spring, а не отдельный фреймворк, написанный на языке, чуждом многим корпоративным разработчикам.

Более подробную информацию о Spring Roo можно найти по адресу: <http://www.springsource.org/roo>.

Расширения для Spring

Помимо проектов, описанных выше, существует также коллекция расширений для Spring, поддерживаемых сообществом, найти которую можно по адресу: <http://www.springsource.org/extensions>. В их числе можно назвать следующие:

- ❑ реализация Spring для языка Python;
- ❑ хранилище для больших двоичных объектов (BLOB);
- ❑ хранилища данных db4o и CouchDB;
- ❑ библиотека на основе Spring для управления рабочим процессом;
- ❑ расширения Kerberos и SAML для Spring Security.

1.4. Что нового в Spring

Прошло почти три года, как вышло второе издание этой книги, и за эти годы произошло множество разных событий. Вышло два значимых выпуска Spring Framework с новыми особенностями и улучшениями, упрощающими разработку приложений. Существенным изменениям подверглись также некоторые элементы экосистемы Spring.

Многие из этих изменений будут рассматриваться на протяжении всей книги. Тем не менее коротко познакомимся, что нового появилось в фреймворке.

1.4.1. Что нового в Spring 2.5?

В ноябре 2007 команда Spring выпустила версию 2.5 фреймворка Spring Framework. Заметным новшеством в Spring 2.5 стала поддержка разработки на основе аннотаций. До появления версии Spring 2.5

нормой считалось определение конфигураций в виде XML-файлов. Но в версии Spring 2.5 появилось несколько дополнительных способов использования аннотаций, существенно уменьшивших объем XML-кода, необходимого для конфигурирования Spring:

- ❑ внедрение зависимостей посредством аннотации `@Autowired` и управление автоматическим связыванием с помощью аннотаций `@Qualifier`;
- ❑ поддержка аннотаций JSR-250, включая `@Resource` – для внедрения зависимости от именованного ресурса, а также `@PostConstruct` и `@PreDestroy` – для реализации методов управления жизненным циклом;
- ❑ автоматическое определение компонентов Spring, отмеченных аннотацией `@Component` (или одной из нескольких стереотипных аннотаций);
- ❑ совершенно новая модель программирования Spring MVC с применением аннотаций, которая существенно упростила разработку веб-приложений на основе Spring;
- ❑ новый фреймворк для интеграционного тестирования, основанный на JUnit 4 и аннотациях.

Даже при том, что аннотации стали самым заметным новшеством в Spring 2.5, тем не менее перечень нововведений на этом не ограничивается:

- ❑ полная поддержка Java 6 и Java EE 5, включая JDBC 4.0, JTA 1.1, JavaMail 1.4 и JAX-WS 2.0;
- ❑ новое выражение `bean-name` в элементе, описывающем точку внедрения, для внедрения аспектов в компоненты Spring по их именам;
- ❑ встроенная поддержка AspectJ для этапа загрузки приложения в JVM;
- ❑ новые пространства имен XML для определения конфигураций, включая пространство имен `context` – для конфигурирования особенностей контекста, и пространство имен `jms` – для конфигурирования компонентов, управляемых сообщениями;
- ❑ поддержка именованных параметров в `SqlJdbcTemplate`.

Многие из этих новых особенностей фреймворка Spring мы будем исследовать на протяжении всей книги.

1.4.2. Что нового в Spring 3.0?

После появления всех этих замечательных особенностей в Spring 2.5 сложно представить, что могло бы появиться в Spring 3.0. Но

в версии 3.0 фреймворк Spring превзошел самого себя в области использования аннотаций и получил несколько новых особенностей:

- ❑ Полноценная поддержка REST в Spring MVC, включая контроллеры Spring MVC, возвращающие данные в формате XML, JSON, RSS и др., при обращении к URL-адресам, оформленным в стиле REST. Новая поддержка REST в Spring 3 будет рассматриваться в главе 12.
- ❑ Новый язык выражений, поднявший прием внедрения зависимостей в Spring на новый уровень, обеспечив возможность внедрения значений из различных источников, включая другие компоненты и системные свойства. Подробнее язык выражений в Spring будет рассматриваться в следующей главе.
- ❑ Новые аннотации для Spring MVC, включая `@CookieValue` и `@RequestHeader`, позволяющие извлекать значения из cookies заголовков запросов соответственно. Как использовать эти аннотации, будет показано в главе 8, при обсуждении Spring MVC.
- ❑ Новое пространство имен XML для упрощения конфигурирования Spring MVC.
- ❑ Поддержка декларативных проверок с помощью аннотаций JSR-303 (Bean Validation API).
- ❑ Поддержка новой спецификации JSR-330 внедрения зависимостей.
- ❑ Применение аннотаций для объявления асинхронных методов и методов, вызываемых по расписанию.
- ❑ Новая модель конфигурирования на основе аннотаций, позволяющая выполнять конфигурирование фреймворка Spring практически без применения XML. Этот новый способ конфигурирования будет рассматриваться в следующей главе.
- ❑ Механизм отображения объектов в XML-формат (Object-to-XML Mapping, OXM) из проекта Spring Web Services был перенесен в основной фреймворк Spring Framework.

Помимо новых особенностей, появившихся в версии Spring 3.0, также важно упомянуть, что было исключено из Spring 3.0. В частности, начиная с версии Spring 3.0 фреймворк может выполняться только под управлением Java 5, так как Java 1.4 достигла конца своего жизненного пути и больше не будет поддерживаться в Spring.

1.4.3. Что нового в экосистеме Spring?

Помимо основного фреймворка Spring Framework, проекты, основанные на Spring, также могут похвастаться впечатляющими ново-

введениями. В книге не так много места, чтобы подробно охватить все изменения, однако есть несколько пунктов, достаточно важных, чтобы упомянуть их:

- ❑ Вышла версия Spring Web Flow 2.0, поддерживающая упрощенную схему определения последовательностей операций, что еще больше упрощает создание диалоговых веб-приложений.
- ❑ Вместе с версией Spring Web Flow 2.0 появились *Spring JavaScript* и *Spring Faces*. Spring JavaScript – это JavaScript-библиотека, обеспечивающая возможность придания веб-страницам динамического поведения. Spring Faces – это фреймворк, позволяющий использовать технологию JSF совместно с фреймворками Spring MVC и Spring Web Flow.
- ❑ Старый фреймворк Acegi Security был полностью переработан и выпущен под названием Spring Security 2.0. Новая инкарнация Spring Security предлагает новую схему конфигурирования, которая существенно уменьшает объем кода разметки XML, необходимого для настройки безопасности приложения.

Как раз когда я писал эту книгу, фреймворк продолжал развиваться. В результате недавно была выпущена версия Spring Security 3.0, еще более упрощающая декларативную поддержку безопасности за счет использования преимуществ нового языка выражений в Spring для объявления констант.

Как видите, проект Spring активно продолжает развиваться. В нем постоянно появляется что-то новое, еще больше упрощающее разработку корпоративных приложений на языке Java.

1.5. В заключение

Теперь вы должны иметь неплохое представление, какими возможностями обладает Spring. Основная цель фреймворка Spring – сделать разработку корпоративных приложений на языке Java как можно проще и способствовать слабой связанности кода. Наиболее важную роль в этом играют внедрение зависимостей и аспектно-ориентированное программирование.

В этой главе вы познакомились с реализацией механизма внедрения зависимостей в Spring. DI – это такой способ связывания прикладных объектов, что им не требуется знать, откуда проистекают их зависимости и как они реализованы. Вместо самостоятельного приобретения своих зависимостей в зависимые объекты извне внедряются ссылки на объекты, от которых они зависят. Поскольку



зависимые объекты взаимодействуют с внедренными объектами только через их интерфейсы, связь между ними остается слабой.

В дополнение к механизму внедрения зависимостей мы также увидели блестящую поддержку аспектно-ориентированного программирования в Spring. AOP позволяет сконцентрировать логику, которая обычно бывает разбросана по всему приложению, в одном месте – аспекте. Когда Spring связывает компоненты друг с другом, такие аспекты могут быть вплетены в них во время выполнения, фактически добавляя в компоненты новые черты поведения.

Внедрение зависимостей и AOP занимает центральное положение в Spring. Поэтому вы должны понимать, как использовать эти основные функции, чтобы иметь возможность задействовать оставшуюся часть фреймворка. В этой главе мы лишь слегка коснулись особенностей DI и AOP. В следующих нескольких главах мы углубимся в изучение DI и AOP. А теперь перейдем к главе 2, чтобы узнать, как связываются объекты в Spring с помощью механизма внедрения зависимостей.



Глава 2. Связывание компонентов

В этой главе рассматриваются следующие темы:

- объявление компонентов;
- внедрение через конструкторы и методы доступа;
- связывание компонентов;
- управление созданием и уничтожением компонентов.

Доводилось ли вам когда-нибудь задерживаться после просмотра фильма, чтобы посмотреть титры? Это невероятно, сколько людей требуется сплотить для создания крупной кинокартины. Помимо очевидных участников – актеров, сценаристов, режиссеров и продюсеров, в создании фильма участвуют музыканты, специалисты по спецэффектам, художественные руководители. И это не говоря уже о главном постановщике, звукорежиссере, костюмерах, гримерах, постановщиках трюков, публицистах, первом ассистенте оператора, втором ассистенте оператора, художнике-декораторе, главном осветителе и (возможно, самых важных) поварах.

Теперь представьте, на что был бы похож фильм, если бы эти люди не общались друг с другом. Скажем, если бы все они пришли в студию и начали делать свою работу без какой-либо координации. Если режиссер остается на месте и не говорит «камера, съемка», оператор не начинает снимать. Наверное, это все равно не будет иметь значения, так как главная актриса будет оставаться в своем трейлере и свет не будет работать, потому что главный осветитель не будет нанят. Возможно, вам доводилось видеть фильм, где все так и было. Однако большинство фильмов (во всяком случае, хороших) – это результат труда сотен людей, работающих вместе ради общей цели – создания кассового фильма.

В этом отношении большая часть программного обеспечения не является исключением. Любое нетривиальное приложение состоит из нескольких объектов, которые действуют совместно для достижения некоторой цели. Эти объекты должны знать друг о друге и



взаимодействовать друг с другом для выполнения своей работы. В приложении интернет-магазина, например, компоненту, управляющему заказом, может потребоваться взаимодействовать с компонентом, управляющим товаром, и компонентом, управляющим доступом к кредитной карте. Всем им наверняка понадобится работать с компонентом доступа к базе данных.

Однако, как было показано в главе 1, традиционные подходы к созданию связей между прикладными объектами (через вызов конструктора или поиск) усложняют программный код, который трудно использовать повторно и тестировать. В лучшем случае эти объекты будут делать больше работы, чем должны. В худшем – они будут сильно связаны между собой и станут слишком сложными для повторного использования и тестирования.

В Spring объекты не отвечают за поиск или создание других объектов, необходимых для выполнения работы. Вместо этого контейнер передает им ссылки на объекты, с которыми они должны взаимодействовать. Компоненту, управляющему заказом, например, может понадобиться компонент доступа к кредитной карте, однако он не должен создавать компонент доступа к кредитной карте. Ему надо только протянуть руку – и он получит требуемый компонент.

Создание взаимосвязей между прикладными объектами является сущностью технологии внедрения зависимостей (DI) и обычно называется *связыванием* (wiring). В этой главе мы рассмотрим основы связывания компонентов с помощью фреймворка Spring. Так как DI является одной из основ Spring, этот прием вы будете использовать практически постоянно при разработке приложений на основе Spring.

2.1. Объявление компонентов

А теперь я приглашаю вас на первый (и, возможно, последний) ежегодный конкурс талантов JavaBean. Я обследовал страну (фактически только рабочее пространство IDE) в поисках лучших из компонентов JavaBeans, которые будут выступать здесь и в следующих нескольких главах. Мы устроим состязание, а наши судьи поставят оценки. Итак, Spring-программисты, открываем конкурс «Spring Idol».

Для участия в конкурсе нам потребуется несколько исполнителей, которые определены интерфейсом `Performer`:

```
package com.springaction.springidol;

public interface Performer {
    void perform() throws PerformanceException;
}
```

В конкурсе талантов «Spring Idol» вы встретитесь с несколькими участниками, каждый из которых реализует интерфейс `Performer`. Но для начала подготовим сцену, рассмотрев базовую конфигурацию Spring.

2.1.1. Подготовка конфигурации Spring

Как уже отмечалось, основу фреймворка Spring составляет *контейнер*. Если не выполнить настройку Spring, у нас в руках окажется пустой контейнер, не имеющий никакой практической ценности. Поэтому необходимо сконфигурировать фреймворк Spring, чтобы сообщить ему, какие компоненты должны находиться в контейнере и как они должны быть связаны между собой.

Начиная с версии Spring 3.0, существуют два пути настройки компонентов в контейнере Spring. Первый путь – традиционный, когда конфигурация описывается в одном или более XML-файлах. Но версия Spring 3.0 предлагает еще один путь, основанный на программном коде Java. Сейчас мы рассмотрим традиционный способ, опирающийся на XML-файлы, а новый способ, основанный на программном коде Java, мы рассмотрим в разделе 4.4.

При объявлении компонентов в XML-файле роль корневого элемента конфигурационного файла играет элемент `<beans>` из схемы `beans`. Ниже показано, как выглядит типичный XML-файл с конфигурацией фреймворка Spring:

```
<?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-3.0.xsd">

    <!-- Здесь должны находиться объявления компонентов -->

</beans>
```

Внутрь элемента `<beans>` можно поместить полное описание конфигурации Spring, включая объявления `<bean>`. Однако пространство



имен beans не является единственным, поддерживаемым фреймворком Spring. Всего основной фреймворк Spring Framework поддерживает десять пространств имен для описания конфигураций, перечисленных в табл. 2.1.

Таблица 2.1. Фреймворк Spring поддерживает несколько пространств имен XML, посредством которых выполняется настройка контейнера Spring

Пространство имен	Назначение
aop	Предоставляет элементы для объявления аспектов и для автоматического проксирования классов, объявляемых аспектами с помощью аннотации @AspectJ
beans	Пространство имен с основными элементами, позволяющими объявлять компоненты и определять связи между ними
context	Содержит элементы для конфигурирования контекста приложения Spring, включая возможность автоматического определения и автоматического связывания компонентов, а также внедрения объектов, которые не управляются фреймворком Spring непосредственно
jee	Обеспечивает интеграцию с такими Java EE API, как JNDI и EJB
jms	Предоставляет элементы для объявления POJO, управляемых сообщениями
lang	Позволяет объявлять компоненты, реализованные как сценарии на языках Groovy, JRuby и BeanShell
mvc	Включает такие возможности Spring MVC, как аннотированные контроллеры, контроллеры представлений и обработчики
oxm	Поддерживает возможность настройки механизма отображения объектов в XML
tx	Содержит элементы настройки декларативных транзакций
util	Набор различных вспомогательных элементов, включающий возможность объявления коллекций как компонентов и поддержку элементов-заполнителей свойств

Помимо пространств имен, поддерживаемых основным фреймворком Spring Framework, многие проекты из экосистемы Spring, такие как Spring Security, Spring Web Flow и Spring Dynamic Modules, добавляют собственные пространства имен для определения конфигураций.

С дополнительными пространствами имен мы будем знакомиться на протяжении всей книги. А пока заполним пустующее пространство в конфигурационном файле, добавив несколько элементов <bean> внутри элемента <beans>.

2.1.2. Объявление простого компонента

В отличие от некоторых конкурсов талантов с аналогичными названиями, о которых вы могли слышать, «Spring Idol» учитывает не только певцов. На самом деле многим исполнителям медведь на ухо наступил. Например, один из исполнителей – жонглер, представленный в листинге 2.1.

Листинг 2.1. Компонент-жонглер

```
package com.springinaction.springidol;

public class Juggler implements Performer {
    private int beanBags = 3;

    public Juggler() {
    }

    public Juggler(int beanBags) {
        this.beanBags = beanBags;
    }

    public void perform() throws PerformanceException {
        System.out.println("JUGGLING " + beanBags + " BEANBAGS");
    }
}
```

Как видите, класс жонглера `Juggler` реализует не только интерфейс `Performer`, посредством которого он сообщает, что жонглирует несколькими мячиками. По умолчанию жонглер жонглирует тремя мячиками, но через конструктор ему можно передать любое другое их количество.

Итак, мы определили класс `Juggler`, а теперь добро пожаловать на сцену, наш первый исполнитель Дюк! Ниже показано, как объявлен Дюк в конфигурационном файле Spring (`spring-idol.xml`):

```
<bean id="duke"
      class="com.springinaction.springidol.Juggler" />
```

Элемент `<bean>` является самым основным в конфигурационном файле Spring. Он предписывает фреймворку Spring создать объект. Здесь мы объявили объект `duke` как компонент, управляемый фреймворком Spring, используя чуть ли не самое простейшее объявление `<bean>`, которое только возможно. Атрибут `id` присваивает компоненту



имя, которое можно использовать для обращения к компоненту в контейнере Spring. Этот компонент будет известен как duke. А атрибут class сообщает фреймворку тип компонента. Компонент duke – это объект класса Juggler.

В процессе загрузки компонентов контейнер Spring создаст компонент duke, используя конструктор по умолчанию. По сути, компонент duke будет создан выполнением следующего программного кода¹:

```
new com.springinaction.springidol.Juggler();
```

Чтобы дать Дюку возможность выступить, можно загрузить контекст приложения, как показано ниже:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(  
    "com/springinaction/springidol/spring-idol.xml");  
  
Performer performer = (Performer) ctx.getBean("duke");  
performer.perform();
```

Несмотря на то что это не настоящий конкурс, предыдущий код дает Дюку шанс выступить. При выполнении он выведет следующее:

```
JUGGLING 3 BEANBAGS
```

По умолчанию Дюк жонглирует только тремя мячиками. Однако жонглирование тремя мячами не выглядит чем-то особенным – это по силам любому. Чтобы Дюк имел хоть какую-то надежду на победу в конкурсе талантов, ему понадобится показать свое умение жонглировать гораздо большим количеством мячиков. Посмотрим, как сконфигурировать Дюка, чтобы он мог стать жонглером-победителем.

2.1.3. Внедрение через конструкторы

Чтобы действительно удивить судей, Дюк решает побить мировой рекорд, жонглируя не меньше чем 15 мячиками одновременно².

¹ Обратите внимание на слова «по сути». В действительности фреймворк Spring создает компоненты, используя механизм рефлексии.

² Дополнительная информация: чтобы установить мировой рекорд по жонглированию мячиками, нужно не только уметь жонглировать большим их количеством, но и жонглировать достаточно долго. Несколько мировых рекордов принадлежит Брюсу Сарафяну (Bruce Sarafian), в том числе

Как было показано в листинге 2.1, класс `Juggler` может быть создан двумя разными способами:

- ❑ вызовом конструктора по умолчанию;
- ❑ вызовом конструктора с аргументом типа `int`, определяющим количество мячиков, которые жонглер `Juggler` должен попытаться удержать в воздухе.

Объявление компонента `duke` в разделе 2.1.2 выполнено правильно, но в нем используется конструктор по умолчанию класса `Juggler`, что позволяет Дюку жонглировать только тремя мячиками. Чтобы сделать Дюка рекордсменом по жонглированию, необходимо использовать другой конструктор. Ниже показано иное объявление Дюка как жонглера 15 мячиками:

```
<bean id="duke"
      class="com.springinaction.springidol.Juggler">
    <constructor-arg value="15" />
</bean>
```

Элемент `<constructor-arg>` здесь используется для передачи фреймворку Spring дополнительной информации, касающейся создания компонента. Если элемент `<constructor-arg>` не указан, как в разделе 2.1.2, будет использоваться конструктор по умолчанию. Но здесь присутствует элемент `<constructor-arg>` со значением 15 в атрибуте `value`, поэтому фреймворт задействует другой конструктор.

Теперь при выступлении Дюка будет выведено следующее:

JUGGLING 15 BEANBAGS

Жонглирование 15 мячиками одновременно должно выглядеть весьма впечатляюще. Но Дюк – не только хороший жонглер, но еще и искусный декламатор. Чтобы научиться жонглировать и одновременно декламировать стихи, необходимо много тренироваться. Если Дюк сможет жонглировать и одновременно читать сонеты Шекспира, он наверняка сможет стать абсолютным победителем конкурса. (Я предупреждал, что это будет необычный конкурс!)

жонглирование 12 мячиками до 12 поимок. Другой рекордсмен, Энтони Гатто (Anthony Gatto), в 2005 году установил рекорд, жонглируя 7 мячиками в течение 10 минут и 12 секунд. Еще один жонглер, Питер Бон (Peter Bone), заявил, что жонглировал 13 мячиками до 13 поимок, но не представил видеозаписи, доказывающей это достижение.

Внедрение ссылок через конструкторы

Так как Дюк – не обычный жонглер, а поэтический жонглер, необходимо определить для него новый тип жонглера. Класс PoeticJuggler (листинг 2.2) точнее описывает таланты Дюка.

Листинг 2.2. Жонглер, декламирующий стихи

```
package com.springinaction.springidol;

public class PoeticJuggler extends Juggler {
    private Poem poem;

    public PoeticJuggler(Poem poem) { // Внедрение поэмы
        super();
        this.poem = poem;
    }

    public PoeticJuggler(int beanBags, Poem poem) { // Внедрение количества
        super(beanBags); // мячиков и поэмы
        this.poem = poem;
    }

    public void perform() throws PerformanceException {
        super.perform();
        System.out.println("While reciting... ");
        poem.recite();
    }
}
```

Этот новый тип жонглера делает все то же, что и обычный жонглер, но также содержит ссылку на декламируемую им поэму. Коль скоро речь зашла о поэме, необходимо определить интерфейс, который описывает поэму:

```
package com.springinaction.springidol;

public interface Poem {
    void recite();
}
```

Один из любимых сонетов Дюка – сонет Шекспира «Когда в раздоре с миром и судьбой»¹. Класс Sonnet29 (листинг 2.3) реализует интерфейс Poem, определяющий этот сонет.

¹ Сонет 29 в переводе С. Я. Маршака. – *Прим. перев.*

Листинг 2.3. Класс, олицетворяющий выдающееся произведение поэта

```
package com.springinaction.springidol;
public class Sonnet29 implements Poem {
    private static String[] LINES = {
        "Когда в раздоре с миром и судьбой,",
        "Припомнив годы, полные невзгод,",
        "Тревожу я бесплодною мольбой",
        "Глухой и равнодушный небосвод",
        "И, жалуясь на горестный удел,",
        "Готов меняться жребием своим",
        "С тем, кто в искусстве больше преуспел,",
        "Богат надеждой и людьми любим, -",
        "Тогда, внезапно вспомнив о тебе,",
        "Я малодушье жалкое кляну,",
        "И жаворонком, вопреки судьбе,",
        "Моя душа несется в вышину.",
        "С твоей любовью, с памятью о ней",
        "Всех королей на свете я сильней."};

    public Sonnet29() {
    }

    public void recite() {
        for (int i = 0; i < LINES.length; i++) {
            System.out.println(LINES[i]);
        }
    }
}
```

Объект класса Sonnet29 можно объявить компонентом Spring, как показано ниже:

```
<bean id="sonnet29"
      class="com.springinaction.springidol.Sonnet29" />
```

Осталось только передать выбранную поэму Дюку. Теперь, когда Дюк стал поэтическим жонглером (`PoeticJuggler`), его описание в элементе `<bean>` следует немного изменить:

```
<bean id="poeticDuke"
      class="com.springinaction.springidol.PoeticJuggler">
    <constructor-arg value="15" />
    <constructor-arg ref="sonnet29" />
</bean>
```

Как показано в листинге 2.2, в этом классе отсутствует конструктор по умолчанию. Единственный способ создать объект PoeticJuggler заключается в том, чтобы использовать конструктор с аргументами. В этом примере используется конструктор, принимающий аргументы типов `int` и `Poem`. В объявлении компонента `duke`, с помощью атрибута `value` элемента `<constructor-arg>`, указывается количество мячиков, равное 15.

Однако использовать атрибут `value` для инициализации второго аргумента конструктора нельзя, потому что `Poem` не является простым типом. Для передачи ссылок на другие компоненты следует использовать атрибут `ref`. В данном случае это должна быть ссылка на компонент с идентификатором `sonnet29`. Контейнер Spring – это не просто механизм создания компонентов. Когда фреймворк Spring встречает определения компонентов `sonnet29` и `duke`, он выполняет дополнительные действия, которые можно выразить следующим программным кодом:

```
Poem sonnet29 = new Sonnet29();
Performer duke = new PoeticJuggler(15, sonnet29);
```

Теперь, когда начнется выступление Дюка, он не только будет жонглировать мячиками, но и прочитает сонет Шекспира, в результате на экран будет выведено следующее:

```
JUGGLING 15 BEANBAGS WHILE RECITING... Когда в раздоре с миром и судьбой,  
Припомнив годы, полные невзгод, Тревожу я бесплодною мольбой Глухой и равнодушный  
небосвод И, жалуясь на горестный удел, Готов меняться жребием своим С тем, кто в  
искусстве больше преуспел, Богат надеждой и людьми любим, - Тогда, внезапно вспом-  
нив о тебе, Я малодущье жалкое кляну, И жаворонком, вопреки судьбе, Моя душа не-  
сется в вышину. С твоей любовью, с памятью о ней Всех королей на свете я сильней.
```

Создание компонентов с применением приема внедрения через конструктор – надежный способ, но как быть, если компонент, который требуется объявить, не имеет общедоступного конструктора? Посмотрим, как реализуется связывание компонентов, создаваемых с помощью фабричных методов.

Создание компонентов с помощью фабричных методов

Иногда единственный способ создать объект заключается в том, чтобы использовать статический *фабричный метод*. Фреймворк Spring

поддерживает возможность связывания компонентов, объявленных с помощью элементов <bean>, которые содержат атрибут factory-method.

Для иллюстрации рассмотрим порядок определения компонента, являющегося экземпляром класса-одиночки¹. Классы-одиночки гарантируют возможность создания только одного экземпляра. Типичным примером класса-одиночки может служить класс Stage, представленный в листинге 2.4 ниже.

Листинг 2.4. Класс-одиночка Stage

```
package com.springinaction.springidol;

public class Stage {
    private Stage() {
    }

    private static class StageSingletonHolder {
        static Stage instance = new Stage(); // Создание экземпляра
    }

    public static Stage getInstance() {
        return StageSingletonHolder.instance; // Возвращает экземпляр
    }
}
```

В конкурсе «Spring Idol» необходимо гарантировать наличие только одной сцены для выступлений. Класс Stage реализован с применением шаблона проектирования «Одиночка» (Singleton), чтобы устранить любые возможности создания более одного его экземпляра.

Однако заметьте, что класс Stage не имеет общедоступного конструктора. Вместо него используется статический метод getInstance(), который каждый раз возвращает один и тот же экземпляр класса. (Для безопасности при выполнении в многопоточной среде создание экземпляра класса в методе getInstance() выполняется с использованием приема, известного как «инициализация по требованию владельца»².) Как же превратить объект Stage, не имеющий общедоступного конструктора, в компонент Spring?

¹ Здесь имеется в виду шаблон проектирования «Одиночка» (Singleton), представленный «бандой четырех», а не понятие определения единичного компонента, существующее в Spring.

² Дополнительную информацию об идиоме «инициализация по требованию владельца» можно найти по адресу: <http://mng.bz/IGYx>.

К счастью, элемент `<bean>` имеет атрибут `factory-method`, позволяющий определить статический метод, который должен вызываться для создания экземпляра класса вместо конструктора. Чтобы превратить объект `Stage` в компонент в контексте Spring, достаточно просто воспользоваться атрибутом `factory-method`, как показано ниже:

```
<bean id="theStage"
      class="com.springinaction.springidol.Stage"
      factory-method="getInstance" />
```

Здесь я показал, как использовать атрибут `factory-method` для создания экземпляра класса-одиночки и преобразования его в компонент Spring, но этот прием подходит для любых случаев, где необходимо связать объект, создаваемый статическим методом. Дополнительные примеры использования атрибута `factory-method` будут представлены в главе 5, где он будет использоваться для получения ссылок на аспекты AspectJ перед их внедрением в зависимости.

2.1.4. Область действия компонента

По умолчанию все компоненты Spring единичны. Когда контейнер передает компонент (либо через связывание, либо как результат вызова метода контейнера `getBean()`), всегда будет передан тот же самый экземпляр компонента. Однако иногда бывает необходимо получить уникальный экземпляр компонента при каждом обращении. Как изменить единичный нрав Spring?

При объявлении `<bean>` компонента можно определить область его действия. Чтобы заставить фреймворк Spring создавать новый экземпляр при каждом обращении, в объявление компонента следует добавить атрибут `scope` со значением `prototype`. Например, представьте, что билеты на выступления объявляются как компоненты Spring:

```
<bean id="ticket"
      class="com.springinaction.springidol.Ticket" scope="prototype" />
```

Очень важно, чтобы каждый посетитель выступлений получил отдельный билет. Если бы существовал только один экземпляр компонента `ticket`, все посетители получали бы один и тот же билет.

Для первого посетителя это не страшно, но всех остальных могли бы обвинить в подделке!

Помимо значения `prototype`, атрибуту `scope` можно присвоить ряд других значений, определяющих область действия компонента, которые перечислены в табл. 2.2.

Таблица 2.2. Фреймворк Spring позволяет определять различные области действия компонентов без жесткого определения правил видимости в программном коде

Область действия	Описание
<code>singleton</code>	В каждом контейнере Spring может быть создан только один компонент (по умолчанию)
<code>prototype</code>	Позволяет создавать произвольное количество компонентов (по одному на каждое обращение)
<code>request</code>	Область действия компонента ограничивается HTTP-запросом. Может применяться только в веб-приложениях Spring (например, использующих Spring MVC)
<code>session</code>	Область действия компонента ограничивается HTTP-сеансом. Может применяться только в веб-приложениях Spring (например, использующих Spring MVC)
<code>global-session</code>	Область действия компонента ограничивается глобальным HTTP-сеансом. Может применяться только в портлетах

В основном вполне достаточно будет оставить область действия в значении по умолчанию `singleton`, но в ситуациях, когда желательно использовать Spring как фабрику новых экземпляров доменных объектов, может пригодиться область действия `prototype`. Если доменные объекты определены как прототипы компонентов, вы легко сможете настроить их в Spring как любые другие компоненты. При этом фреймворк Spring всегда будет возвращать уникальный экземпляр при обращении к прототипу компонента.

Проницательный читатель заметит, что понятие единичных компонентов ограничено областью действия контекста Spring. В отличие от истинных классов-одиночек, гарантирующих существование единственного экземпляра на каждый загрузчик классов (`classloader`), для единичных компонентов в Spring гарантируется только наличие единственного экземпляра компонента в контексте приложения – ничто не мешает создать экземпляр того же класса традиционным способом или даже создать несколько объявлений `<bean>` для одного и того же класса.



2.1.5. Инициализация и уничтожение компонентов

При создании компонента может потребоваться выполнить некоторые операции по его инициализации, чтобы привести его в пригодное для использования состояние. Аналогично, перед уничтожением и удалением компонента из контейнера, может потребоваться выполнить некоторые заключительные операции. Для этой цели фреймворк Spring позволяет определять методы управления жизненным циклом компонента.

Чтобы определить методы, вызываемые при создании и уничтожении компонента, просто добавьте в элемент `<bean>` атрибуты `init-method` и/или `destroy-method`. Атрибут `init-method` определяет метод, вызываемый сразу после создания экземпляра компонента. Аналогично атрибут `destroy-method` определяет метод, вызываемый непосредственно перед удалением компонента из контейнера.

Для иллюстрации представим, что у нас имеется Java-класс `Auditorium`, представляющий концертный зал, где проводится конкурс талантов. Вероятно, класс `Auditorium` должен иметь массу особенностей, но пока сосредоточимся на двух из них, играющих важную роль в начале и в конце выступлений: включение и выключение освещения.

Для поддержки этих важных операций в классе `Auditorium` могутиться методы `turnOnLights()` и `turnOffLights()`:

```
public class Auditorium {  
    public void turnOnLights() {  
        ...  
    }  
  
    public void turnOffLights() {  
        ...  
    }  
}
```

Детали реализации методов `turnOnLights()` и `turnOffLights()` не имеют большого значения. Но большое значение имеет тот факт, что метод `turnOnLights()` должен вызываться перед выступлениями, а метод `turnOffLights()` – по их окончании. Для решения этой задачи добавим атрибуты `init-method` и `destroy-method` в объявление компонента `auditorium`:

```
<bean id="auditorium"
      class="com.springinaction.springidol.Auditorium"
      init-method="turnOnLights"
      destroy-method="turnOffLights"/>
```

В соответствии с этим объявлением сразу после создания экземпляра компонента `auditorium` будет вызван метод `turnOnLights()`, включающий освещение зала. А непосредственно перед удалением компонента из контейнера будет вызван метод `turnOffLights()`, выключающий освещение.

Интерфейсы `InitializingBean` и `DisposableBean`. Фреймворк Spring обеспечивает альтернативный способ определения методов инициализации и уничтожения компонентов – реализовать в классе компонента интерфейсы `InitializingBean` и `DisposableBean`. Компонентам, реализующим эти интерфейсы, предоставляется возможность управлять этапами своего жизненного цикла, и они обрабатываются фреймворком Spring особым образом. Интерфейс `InitializingBean` объявляет метод `afterPropertiesSet()`, который играет роль метода инициализации. А интерфейс `DisposableBean` объявляет метод `destroy()`, который вызывается перед удалением компонента из контекста приложения.

Главным преимуществом использования этих интерфейсов управления жизненным циклом является автоматическое определение контейнером Spring компонентов, реализующих эти интерфейсы, без каких-либо внешних настроек. А недостаток состоит в образовании тесной связи между прикладными компонентами и Spring API. Именно по этой причине для реализации операций инициализации и уничтожения компонентов я рекомендую использовать атрибуты `init-method` и `destroy-method`. Единственное, где могут пригодиться интерфейсы управления жизненным циклом, – при разработке компонентов собственного фреймворка, которые будут использоваться исключительно в контейнере Spring.

Методы инициализации и уничтожения по умолчанию

Если в файле определения контекста необходимо определить множество компонентов с методами *инициализации* или *уничтожения*, имеющими одинаковые имена, можно не объявлять атрибуты `init-method` и `destroy-method` для каждого отдельного компонента, а добавить атрибуты `default-init-method` и `default-destroy-method` в элемент `<beans>`:

```
<?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-3.0.xsd"  
default-init-method="turnOnLights"  
default-destroy-method="turnOffLights">> ...  
</beans>
```

Атрибут default-init-method определяет метод инициализации для всех компонентов в данном определении контекста. Аналогично атрибут default-destroy-method определяет общий метод уничтожения для всех компонентов в определении контекста. В данном случае фреймворк Spring предписывается инициализировать все компоненты, описанные в файле определения контекста, вызовом метода turnOnLights() и уничтожать их вызовом метода turnOffLights() (если эти методы существуют – иначе ничего не произойдет).

2.2. Внедрение в свойства компонентов

Как правило, свойства компонентов JavaBean являются частными и имеют пару методов доступа с именами вида setXXX() и getXXX(). Фреймворк Spring может использовать метод set для настройки его значения посредством внедрения через метод записи.

Для демонстрации другой формы DI пригласим на сцену нашего следующего участника. Кенни (Kenny) – талантливый музыкант, как определено классом Instrumentalist в листинге 2.5.

Листинг 2.5. Определение исполнителя, который является талантливым музыкантом

```
package com.springinaction.springidol;  
  
public class Instrumentalist implements Performer {  
    public Instrumentalist() {  
    }  
  
    public void perform() throws PerformanceException {  
        System.out.print("Playing " + song + " : ");  
        instrument.play();  
    }  
  
    private String song;  
  
    public void setSong(String song) { // Внедрение мелодии  
        this.song = song;  
    }
```

```
}

public String getSong() {
    return song;
}

public String screamSong() {
    return song;
}

private Instrument instrument;

public void setInstrument(Instrument instrument) { // Внедрение
    this.instrument = instrument; // инструмента
}
}
```

Как показано в листинге 2.5, класс `Instrumentalist` содержит два свойства: `song` и `instrument`. Свойство `song` хранит название мелодии, которую музыкант будет исполнять, и используется в методе `perform()`. Свойство `instrument` хранит ссылку на экземпляр класса `Instrument`, инструмента, на котором музыкант будет играть. Класс `Instrument` имеет следующий интерфейс:

```
package com.springinaction.springidol;

public interface Instrument {
    public void play();
}
```

Поскольку класс `Instrumentalist` имеет конструктор по умолчанию, конкурсанта Кенни можно объявить компонентом `<bean>`, как показано ниже:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist" />
```

У фреймворка Spring не возникнет проблемы с созданием компонента `kenny` как экземпляра класса `Instrumentalist`, однако в этом случае Кенни придется несладко – он будет вынужден выступать, не имея ни мелодии, ни инструмента. Посмотрим, как можно передать Кенни мелодию и инструмент с использованием приема внедрения через методы записи.

2.2.1. Внедрение простых значений

Свойства компонента могут быть настроены в Spring с помощью элемента `<property>`. Элемент `<property>` во многом схож с элементом `<constructor-arg>`, за исключением того, что вместо внедрения значений через аргументы конструктора элемент `<property>` вызывает метод записи свойства.

Для иллюстрации передадим Кенни мелодию для исполнения с помощью приема внедрения через метод записи. Следующий фрагмент XML представляет доработанное объявление компонента `kenny`:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
</bean>
```

Как только будет создан экземпляр класса `Instrumentalist`, Spring воспользуется методами записи, чтобы присвоить указанные значения свойствам, описанным элементами `<property>`. Элемент `<property>` в этом фрагменте XML предписывает фреймворку Spring вызвать метод `setSong()` для записи значения "Jingle Bells" в свойство `song`.

В данном случае для внедрения строкового значения в свойство используется атрибут `value` элемента `<property>`. Но элемент `<property>` позволяет внедрять не только строковые значения. В атрибуте `value` можно также указывать числовые (`int`, `float`, `java.lang.Double` и другие) и логические значения.

Например, представим, что класс `Instrumentalist` имеет свойство `age` типа `int`, определяющее возраст музыканта. Тогда можно было бы указать возраст Кенни, как показано ниже:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="age" value="37" />
</bean>
```

Обратите внимание, что при определении числового значения атрибут `value` используется точно так же, как при определении строкового значения. Фреймворк Spring автоматически определяет тип значения, опираясь на тип свойства. Поскольку свойство `age` имеет тип `int`, Spring знает, что строку "37" следует преобразовать в целочисленное значение перед вызовом метода `setAge()`.

Использование элемента `<property>` прекрасно подходит для настройки простых свойств компонента, но DI – это больше, чем присваивание жестко определенных значений. Настоящая ценность DI заключается в возможности связывания взаимодействующих объектов, чтобы они не связывали себя сами. С этой целью рассмотрим, как передать Кенни инструмент, на котором он будет играть.

2.2.2. Внедрение ссылок на другие компоненты

Кенни – очень талантливый музыкант и может играть практически на любых инструментах. Если инструмент реализует интерфейс `Instrument`, Кенни сможет извлекать из него музыку. Естественно, у Кенни есть любимый инструмент, саксофон, который определен классом `Saxophone` в листинге 2.6.

Листинг 2.6 Класс `Saxophone` реализует интерфейс `Instrument`

```
package com.springinaction.springidol;

public class Saxophone implements Instrument {
    public Saxophone() {
    }

    public void play() {
        System.out.println("TOOT TOOT TOOT");
    }
}
```

Прежде чем позволить Кенни играть на саксофоне, необходимо объявить его компонентом Spring, как показано ниже:

```
<bean id="saxophone"
      class="com.springinaction.springidol.Saxophone" />
```

Обратите внимание, что в классе `Saxophone` нет свойств, нуждающихся в настройке. Следовательно, в компоненте `saxophone` не требуется использовать элемент `<property>`.

После объявления компонента саксофона его можно передать Кенни. Ниже представлено измененное объявление компонента `kenny`, где для определения свойства `instrument` используется прием внедрения через метод записи:

```
<bean id="kenny2"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

Теперь компонент `kenny` снабжен всем необходимым и Кенни готов к исполнению. Как и с Дюком, мы должны сообщить Кенни о начале выступления следующим Java-кодом (возможно в методе `main()`):

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    "com/springinaction/springidol/spring-idol.xml");
Performer performer = (Performer) ctx.getBean("kenny");
performer.perform();
```

Это не тот программный код, что начнет конкурс «Spring Idol», но он даст Кенни шанс попрактиковаться. Когда он будет выполнен, на экране появится следующее:

```
Playing Jingle Bells : TOOT TOOT TOOT
```

Однако он иллюстрирует важное понятие. Если сравнить этот код с кодом, который заставил выступать Дюка, можно обнаружить, что они мало чем отличаются. В сущности, отличие заключается только в имени компонента. И там, и там используется один и тот же код, несмотря на то что один заставляет выступать жонглера, а другой – музыканта.

Это не столько особенность Spring, сколько преимущество использования интерфейсов. Ссылаясь на исполнителя как на объект, реализующий интерфейс `Performer`, можно не глядя заставить выступать любого конкурсанта, будь то жонглер-декламатор или саксофонист. Именно поэтому фреймворк Spring поощряет использование интерфейсов. И, как вы наверняка успели заметить, интерфейсы рука об руку с DI обеспечивают слабую связаннысть объектов.

Как уже упоминалось, Кенни может играть практически на любом инструменте, при условии что он реализует интерфейс `Instrument`. Несмотря на то что Кенни предпочитает саксофон, можно также попросить его сыграть на фортепиано. Возьмем в качестве примера класс `Piano`, представленный в листинге 2.7.

Листинг 2.7. Класс Piano, реализующий интерфейс Instrument

```
package com.springinaction.springidol;

public class Piano implements Instrument {
    public Piano() {
    }

    public void play() {
        System.out.println("PLINK PLINK PLINK");
    }
}
```

Класс Piano можно объявить компонентом Spring, как показано ниже:

```
<bean id="piano"
      class="com.springinaction.springidol.Piano" />
```

Теперь, после объявления компонента piano, чтобы поменять инструмент, Кенни достаточно просто изменить объявление компонента kenny, как показано ниже:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="piano" />
</bean>
```

После внесения этих изменений Кенни будет играть на фортепиано вместо саксофона. Однако, так как класс Instrumentalist использует свое свойство instrument только через методы интерфейса Instrument, нам не пришлось ничего менять в классе Instrumentalist, чтобы обеспечить поддержку новой реализации класса Instrument. Несмотря на то что музыкант (Instrumentalist) способен играть и на саксофоне (Saxophone), и на фортепиано (Piano), он не имеет тесной связи ни с тем, ни с другим инструментом. Если Кенни решит выбрать цимбалы, достаточно будет просто создать класс HammeredDulcimer и изменить свойство instrument в объявлении компонента kenny.

Внедрение внутренних компонентов

Выше было показано, что Кенни может играть и на саксофоне, и на фортепиано, и на любом другом инструменте, реализующем ин-

терфейс `Instrument`. Но правда также в том, что саксофон и фортепиано могут также использоваться любым другим музыкантом, достаточно лишь внедрить нужный инструмент в его свойство `instrument`. То есть не только Кенни сможет играть на любом инструменте, но и любой другой музыкант (`Instrumentalist`) сможет играть, например, на саксофоне. Фактически для компонентов вполне свойственно быть совместно используемыми другими компонентами.

Проблема, однако, состоит в том, что Кенни немного обеспокоен гигиеническими последствиями от совместного использования своего саксофона с другими музыкантами. Он скорее предпочел бы иметь свой личный саксофон. Чтобы помочь Кенни избежать заражения чужими микробами, мы воспользуемся технологией Spring, известной как *внутренние компоненты*.

Как Java-разработчик вы, вероятно, хорошо знакомы с понятием внутренних классов – классов, которые определены в пределах других классов. Аналогично и внутренние компоненты – это компоненты, которые определяются внутри других компонентов. Для иллюстрации рассмотрим новую конфигурацию компонента `kenny`, где его саксофон объявляется как внутренний компонент:

```
<bean id="kenny"
    class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument">
        <bean class="org.springinaction.springidol.Saxophone" />
    </property>
</bean>
```

Как видите, внутренний компонент определяется с помощью элемента `<bean>`, вложенного непосредственно в элемент `<property>` свойства, куда он должен быть внедрен. В данном случае будет создан объект `Saxophone` и внедрен в свойство `instrument` компонента `kenny`.

Внутренние компоненты можно внедрять не только с помощью приема внедрения через методы записи. Внутренние компоненты можно также внедрять через аргументы конструктора, как показано в следующем новом объявлении компонента `duke`:

```
<bean id="duke"
    class="com.springinaction.springidol.PoeticJuggler">
    <constructor-arg value="15" />
    <constructor-arg>
```

```
<bean class="com.springinaction.springidol.Sonnet29" />
</constructor-arg>
</bean>
```

Здесь экземпляр класса Sonnet29 будет создан как внутренний компонент и передан в виде аргумента конструктору класса PoeticJuggler.

Обратите внимание, что внутренние компоненты не имеют атрибута `id`. Нет ничего противозаконного в том, чтобы добавить атрибут `id` в объявление внутреннего компонента, однако в этом нет необходимости, потому что к внутреннему компоненту никто и никогда не будет обращаться по имени. Это является основным недостатком внутренних компонентов: они не могут быть повторно использованы. Внутренние компоненты могут использоваться только для однократного внедрения, и к ним не могут обращаться другие компоненты.

Можно также заметить, что определение внутреннего компонента оказывает негативное влияние на удобочитаемость конфигурационных XML-файлов.

2.2.3. Связывание свойств с помощью пространства имен `ρ`

Внедрение значений и ссылок в свойства компонентов с помощью элемента `<property>` не представляет большого труда. Тем не менее пространство имен `ρ` фреймворка Spring позволяет использовать иной способ связывания свойства компонентов, не требующий такого большого количества угловых скобок.

Пространство имен `ρ` имеет URI схемы <http://www.springframework.org/schema/ρ>. Для его использования достаточно просто добавить его определение в XML-файл конфигурации Spring:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:p="http://www.springframework.org/schema/p"
       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-3.0.xsd">
```

После этого появится возможность использовать для связывания свойств атрибуты с префиксом `p:` элемента `<bean>`. Например, взгляните на следующее объявление компонента `kenny`:

```
<bean id="kenny" class="com.springinaction.springidol.Instrumentalist"
      p:song = "Jingle Bells"
      p:instrument-ref = "saxophone" />
```

Атрибут `p:song` со значением `"Jingle Bells"` внедряет это значение в свойство `song`. Однако атрибут `p:instrument-ref` со значением `"saxophone"` в действительности внедряет в свойство `instrument` ссылку на компонент с идентификатором `saxophone`. Окончание `-ref` подсказывает фреймворку Spring, что вместо буквального значения он должен внедрить ссылку на соответствующий компонент.

Выбор между элементом `<property>` и пространством имен `p` остается за вами. Они действуют совершенно одинаково. Основное преимущество пространства имен `p` состоит в том, что оно обеспечивает более краткую форму записи. Его особенно удобно использовать при создании примеров для книги, ширина страниц в которой ограничена. Поэтому далее я буду использовать пространство имен `p` время от времени, особенно при нехватке пространства по горизонтали.

На данный момент талант Кенни распространяется практически на любой инструмент. Тем не менее он имеет одно ограничение: Кенни способен одновременно играть только на одном инструменте. Следующим на сцену в конкурсе «Spring Idol» выйдет Хэнк, исполнитель, который может играть сразу на нескольких инструментах.

2.2.4. Внедрение коллекций

До сих пор демонстрировались особенности определения простых значений свойств (с помощью атрибута `value`) и ссылок на другие компоненты (с помощью атрибута `ref`). Однако атрибуты `value` и `ref` можно использовать, только когда свойства компонента содержат единственное значение. А возможно ли с помощью Spring инициализировать свойства, имеющие множество значений, что, если свойство является коллекцией значений?

Фреймворк Spring предлагает четыре типа элементов определения коллекций, которые пригодятся для конфигурирования значений, являющихся коллекциями. В табл. 2.3 приводится список этих элементов и описывается, для чего они предназначены.

Элементы `<list>` и `<set>` можно использовать для настройки свойств, которые являются массивами или одной из реализаций `java.util.Collection`. Как будет показано чуть ниже, фактическая реализация коллекции, используемая для определения свойства,

имеет некоторое значение при выборе между элементами <list> и <set>. Оба элемента могут использоваться почти взаимозаменяющими, со свойствами любого типа java.util.Collection.

Таблица 2.3. Подобно тому как в Java имеется несколько видов коллекций, фреймворк Spring так же позволяет внедрение нескольких видов коллекций

Элемент-коллекция	Назначение
<list>	Связывание списка значений, допускаются повторяющиеся значения
<set>	Связывание множества значений, гарантирует отсутствие повторяющихся значений
<map>	Связывание коллекций пар имя/значение, где имя и значение могут быть значениями любых типов
<props>	Связывание коллекций пар имя/значение, где имя и значение должны иметь строковый тип (String)

Что касается элементов <map> и <props>, они соответствуют коллекциям с интерфейсами java.util.Map и java.util.Properties соответственно. Эти типы коллекций можно использовать, когда требуется коллекция, состоящая из пар ключ/значение. Ключевым отличием между ними состоит в том, что при использовании элемента <props> ключи и значения в коллекции должны быть значениями типа String, тогда как при использовании элемента <map> ключи и значения могут быть любых типов.

А теперь, чтобы продемонстрировать возможность связывания коллекций в Spring, поприветствуем Хэнка на сцене «Spring Idol». Особенность таланта Хэнка в том, что он – человек-оркестр. Как и Кенни, Хэнк способен играть на нескольких инструментах, но Хэнк может играть на нескольких инструментах одновременно. Хэнк определен классом OneManBand, как показано в листинге 2.8.

Листинг 2.8. Исполнитель, который является человеком-оркестром

```
package com.springinaction.springidol;

import java.util.Collection;

public class OneManBand implements Performer {
    public OneManBand() {
    }
}
```

```
public void perform() throws PerformanceException {  
    for (Instrument instrument : instruments) {  
        instrument.play();  
    }  
}  
  
private Collection<Instrument> instruments;  
  
public void setInstruments(Collection<Instrument> instruments) {  
    this.instruments = instruments; // Внедрение коллекции инструментов  
}  
}
```

Как показано в листинге 2.8, во время выступления объект OneManBand выполняет итерации по коллекции инструментов. Наиболее важно здесь, что коллекция инструментов внедряется через метод setInstruments(). Посмотрим, как Spring сможет обеспечить Хэнка своей коллекцией инструментов.

Внедрение списков, множеств и массивов

Чтобы передать Хэнку коллекцию инструментов для выступления, воспользуемся элементом <list>:

```
<bean id="hank"  
      class="com.springinaction.springidol.OneManBand">  
  <property name="instruments">  
    <list>  
      <ref bean="guitar" />  
      <ref bean="cymbal" />  
      <ref bean="harmonica" />  
    </list>  
  </property>  
</bean>
```

Элемент <list> содержит одно или более значений. Здесь элементы <ref> используются для определения ссылок на другие компоненты в контексте Spring и передают Хэнку гитару, цимбалы и губную гармошку. Однако внутри элемента <list> можно также использовать другие элементы, определяющие значения, включая <value>, <bean> и <null/>. В действительности элемент <list> может содержать в себе другие элементы <list> как элементы многомерных списков.

В листинге 2.8 свойство instruments класса OneManBand имеет тип java.util.Collection, и для сохранения в нем значений типа Instrument используются обобщения Java 5. Но элемент <list> может применяться к свойствам, которые могут быть любыми реализациями java.util.Collection или массивом. Другими словами, элемент <list> можно было бы использовать, даже если свойство instruments было бы объявлено как:

```
java.util.List<Instrument> instruments;
```

или как:

```
Instrument[] instruments;
```

Аналогично для связывания свойств, являющихся коллекциями или массивами можно использовать элемент <set>:

```
<bean id="hank"
      class="com.springinaction.springidol.OneManBand">
    <property name="instruments">
      <set>
        <ref bean="guitar" />
        <ref bean="cymbal" />
        <ref bean="harmonica" />
        <ref bean="harmonica" />
      </set>
    </property>
</bean>
```

Как уже говорилось, элементы <list> и <set> могут использоваться для связывания значений с любой реализацией java.util.Collection или массивом. Если свойство имеет тип java.util.Set, это еще не означает, что для его инициализации следует обязательно использовать элемент <set>. Даже при том, что возможность инициализировать свойство типа java.util.List с помощью элемента <set> может показаться странной, в действительности это вполне допустимо. В этом случае гарантируется уникальность элементов списка List.

Связывание отображений

Когда выступает человек-оркестр (OneManBand), звук каждого инструмента выводится методом perform() в процессе выполнения итераций по коллекции инструментов. Но предположим, что мы также

хотим видеть, какой инструмент соответствует каждому извлекаемому звуку. Чтобы реализовать это, изменим класс OneManBand, как показано в листинге 2.9.

Листинг 2.9. Преобразование коллекции инструментов класса OneManBand в отображение (Map)

```
package com.springinaction.springidol;

import java.util.Map;
import com.springinaction.springidol.Instrument;
import com.springinaction.springidol.PerformanceException;
import com.springinaction.springidol.Performer;

public class OneManBand implements Performer {
    public OneManBand() {
    }

    public void perform() throws PerformanceException {
        for (String key : instruments.keySet()) {
            System.out.print(key + " : ");
            Instrument instrument = instruments.get(key);
            instrument.play();
        }
    }

    private Map<String, Instrument> instruments;

    public void setInstruments(Map<String, Instrument> instruments) {
        this.instruments = instruments; // Внедрение инструментов в виде
                                         // отображения (Map)
    }
}
```

В новой версии класса OneManBand свойство instruments имеет тип отображения java.util.Map, каждый член которого имеет ключ типа String и значение типа Instrument. Так как члены отображения представлены парами ключ/значение, простых элементов <list> или <set> будет недостаточно для связывания свойства.

Вместо этого для конфигурирования свойства instruments в следующем объявлении компонента hank используется элемент <map>:

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
    <property name="instruments">
        <map>
            <entry key="GUITAR" value-ref="guitar" />
```

```

<entry key="CYMBAL" value-ref="cymbal" />
<entry key="HARMONICA" value-ref="harmonica" />
</map>
</property>
</bean>

```

Элемент `<map>` объявляет значение типа `java.util.Map`. Каждый элемент `<entry>` определяет один элемент отображения. В предыдущем примере атрибут `key` определяет ключ элемента отображения, а атрибут `value-ref` – значение элемента отображения как ссылку на другой компонент внутри контекста Spring.

В нашем примере атрибут `key` используется для определения строкового ключа, а атрибут `value-ref` – для определения ссылочного значения, однако в действительности каждый элемент `<entry>` имеет по два атрибута для определения ключа и значения элемента отображения. Эти атрибуты перечислены в табл. 2.4.

Элемент `<map>` – это единственный способ внедрения пар ключ/значение в свойства компонента, когда один из объектов не является строкой. Посмотрим, как использовать элемент `<props>` для настройки отображения строка в строку.

Таблица 2.4. Элемент `<entry>` в элементе `<map>` позволяет определить ключ и значение элемента отображения, каждый из которых может быть простым значением или ссылкой на другой компонент

Атрибут	Назначение
<code>key</code>	Определяет ключ элемента отображения как строку
<code>key-ref</code>	Определяет ключ элемента отображения как ссылку на компонент в контексте Spring
<code>value</code>	Определяет значение элемента отображения как строку
<code>value-ref</code>	Определяет значение элемента отображения как ссылку на компонент в контексте Spring

Внедрение свойств-коллекций

При описании коллекции Map значений для свойства `instrument` в объекте `OneManBand` значение в каждом элементе отображения было определено с помощью атрибута `value-ref`. Это обусловлено тем, что каждый элемент отображения в конечном счете является другим компонентом в контексте Spring.

Но если при формировании отображения обнаружится, что ключи и значения в нем являются строками, возможно, предпочтительнее

будет вместо класса Map использовать интерфейс java.util.Properties. Интерфейс Properties служит примерно той же цели, что и класс Map, но ограничивает ключи и значения строковым типом.

Для иллюстрации представьте, что вместо коллекции строк и ссылок на компоненты в свойство объекта OneManBand внедряется отображение строки в строку Java.util.Properties. Новое свойство instruments в этом случае можно было бы изменить, как показано ниже.

```
private Properties instruments;
public void setInstruments(Properties instruments) {
    this.instruments = instruments;
}
```

Чтобы связать звуки, издаваемые инструментами, со свойством instruments, в этом случае можно использовать элемент <props>, как показано в следующем объявлении компонента hank:

```
<bean id="hank" class="com.springinaction.springidol.OneManBand">
    <property name="instruments">
        <props>
            <prop key="GUITAR">STRUM STRUM STRUM</prop>
            <prop key="CYMBAL">CRASH CRASH CRASH</prop>
            <prop key="HARMONICA">HUM HUM HUM</prop>
        </props>
    </property>
</bean>
```

Элемент <props> создает отображение типа java.util.Properties, каждый член которого определяется элементом <prop>. Каждый элемент <prop> имеет атрибут key, определяющий ключ члена отображения Properties, а значение определяется содержимым элемента <prop>. В нашем примере элемент с ключом «GUITAR» имеет значение «STRUM STRUM STRUM».

Это, пожалуй, самый сложный для обсуждения элемент конфигурации Spring, потому что термин «property» (свойство) сильно перегружен. Здесь важно придерживаться следующих правил:

- ❑ <property> – это элемент для внедрения значения или ссылки на компонент в «обычное» свойство компонента;
- ❑ <props> – это элемент для определения коллекций типа java.util.Properties;
- ❑ <prop> – это элемент для определения члена коллекции <props>.

Теперь нам известно несколько способов внедрения значений в свойства компонентов. Мы попробовали внедрять простые значения, ссылки на другие компоненты и коллекции. А сейчас посмотрим, как внедрить в свойство компонента пустое значение.

2.2.5. Внедрение пустого значения

Помимо всего прочего, фреймворк Spring может также внедрять в свойства компонентов или аргументы конструкторов пустые значения. Или, если говорить точнее, значение `null`.

Вы, возможно, закатили свои глаза и подумали: «О чём это он? Зачем мне передавать пустое значение в свойство? Разве все свойства не являются пустыми, пока они явно не установлены? В чём подвох?»

Чаще всего дела обстоят именно так – свойства отправляются в путь пустыми и будут оставаться таковыми, пока им не будут присвоены другие значения. Однако некоторые компоненты могут сами присваивать свойствам непустые значения по умолчанию. Что, если по каким-то причинам необходимо, чтобы это свойство получило пустое значение? В этом случае недостаточно просто предполагать, что свойство будет иметь пустое значение, – необходимо явно присвоить ему значение `null`.

Чтобы присвоить свойству значение `null`, достаточно просто воспользоваться элементом `<null/>`. Например:

```
<property name="someNonNullProperty"><null/></property>
```

Другая причина для явного внедрения значения `null` в свойство – отменить автоматическое связывание значения свойства. Что за автоматическое связывание, спросите вы? Продолжайте чтение – мы исследуем автоматическое связывание в главе 4.

А сейчас держитесь крепче за свои кресла. Мы закончим эту главу знакомством с одной из самых сногсшибательных особенностей фреймворка Spring – с языком выражений Spring (Spring Expression Language).

2.3. Внедрение выражений

До сих пор все, что внедрялось в свойства компонентов, было определено в XML-файле конфигурации Spring статически. При внедрении названия музыкального произведения в компонент



Instrumentalist это значение было определено на этапе разработки. А когда внедрялись ссылки на другие компоненты, эти ссылки определялись на этапе создания файла конфигурации Spring.

Но как быть, если потребуется внедрить в свойства значения, которые будут известны только после запуска приложения?

В версии Spring 3.0 появилась возможность использовать *язык выражений Spring* (Spring Expression Language, SpEL) – мощный, но краткий способ внедрения значений в свойства компонентов или аргументы конструкторов с помощью выражений, которые вычисляются на этапе выполнения. С помощью SpEL можно творить чудеса, обрабатывая такие ситуации, которые было бы очень сложно (или даже невозможно) обработать с применением традиционных приемов связывания.

Язык SpEL обеспечивает массу интересных возможностей, среди которых:

- ❑ получение ссылок на компоненты по их идентификаторам;
- ❑ вызов методов и обращение к свойствам объектов;
- ❑ математические и логические операции над значениями, а также операции отношения;
- ❑ сопоставление с регулярными выражениями;
- ❑ операции с коллекциями.

Создание выражений на языке SpEL связано с использованием разнообразных синтаксических конструкций. Даже наиболее замысловатые выражения на языке SpEL зачастую состоят из более простых подвыражений. Но, прежде чем приступить к использованию выражений на языке SpEL, познакомимся с некоторыми основными их ингредиентами.

2.3.1. Основы языка выражений SpEL

Конечная цель выражений на языке SpEL состоит в том, чтобы обеспечить возможность вычисления некоторых значений. В процессе вычислений могут участвовать другие значения. Самой простой разновидностью значений в языке SpEL являются литералы, ссылки на свойства компонентов и константы в некоторых классах.

Литералы

Простейшим выражением на языке SpEL является выражение, состоящее из одного литературного значения. Например, ниже приводится вполне допустимое выражение на языке SpEL:

Неудивительно, что результатом этого выражения является целое число 5. Это значение можно было бы внедрить в свойство компонента с помощью конструкции `#{}()` в атрибуте `value` элемента `<property>`, как показано ниже:

```
<property name="count" value="#{5}" />
```

Конструкция `#{}()` подсказывает фреймворку Spring, что ее содержимое является выражением на языке SpEL. Эти выражения можно также смешивать с обычными значениями:

```
<property name="message" value="The value is #{5}" />
```

На языке SpEL можно также определять вещественные числа. Например:

```
<property name="frequency" value="#{89.7}" />
```

Числа могут записываться в научной форме записи. Как показано в следующем фрагменте, который присваивает значение 10000.0 свойству `capacity`:

```
<property name="capacity" value="#{1e4}" />
```

Язык SpEL также поддерживает строковые литералы, которые могут заключаться в апострофы или кавычки. Например, внедрение строкового литерала в свойство компонента можно записать так:

```
<property name="name" value="#{'Chuck'}" />
```

Если для определения значений XML-атрибутов используются апострофы, выражение на языке SpEL можно заключить в кавычки:

```
<property name='name' value='#{ "Chuck" }' />
```

Для определения логических значений можно использовать пару литералов `true` и `false`. Например, значение `false` можно выразить так:

```
<property name="enabled" value="#{false}" />
```



Работа с литералами в выражениях на языке SpEL является обыденным делом. Однако, чтобы присвоить целочисленному свойству значение 5 или логическому свойству значение `false`, совсем необязательно использовать выражения на языке SpEL. Я считаю, что нет смысла использовать выражения на языке SpEL, содержащие только литералы. Но напомню, что даже самые замысловатые выражения состоят из более простых подвыражений. Поэтому знание приемов работы с литералами не будет лишним. Рано или поздно, с ростом сложности выражений, они все равно потребуются.

Ссылки на компоненты и обращение к их свойствам и методам

Другой основной особенностью выражений на языке SpEL является возможность ссылаться на другие компоненты по их идентификаторам. Например, с помощью выражения на языке SpEL можно внедрить компонент в свойство другого компонента:

```
<property name="instrument" value="#{saxophone}" />
```

Здесь для внедрения компонента с идентификатором `"saxophone"` в свойство `instrument` было использовано выражение SpEL. Но постойте... разве нельзя сделать то же самое без выражения SpEL, воспользовавшись атрибутом `ref`, как показано ниже?

```
<property name="instrument" ref="saxophone"/>
```

Да, результат будет тем же самым. И да, для этого не нужно использовать выражение на языке SpEL. Но весь интерес в том, что это возможно, и чуть ниже я покажу, в каких ситуациях может пригодиться возможность внедрения ссылок на компоненты с помощью выражений. А сейчас я хочу показать, как пользоваться ссылками на компоненты для доступа к их свойствам в выражениях.

Допустим, что необходимо настроить новый компонент `Instrumentalist` с идентификатором `carl`. Карл – необычный музыкант, он – подражатель. Вместо собственного музыкального произведения Карл будет исполнять то же произведение, что исполняет Кенни. При конфигурировании компонента `carl` можно воспользоваться выражением на языке SpEL, чтобы скопировать произведение из свойства `song` Кенни, как показано ниже:

```
<bean id="carl"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="#{kenny.song}" />
</bean>
```

Как показано на рис. 2.1, выражение, внедряющее значение в свойство song, состоит из двух частей.

Первая часть (перед точкой) ссылается на компонент kenny по его идентификатору. Вторая часть ссылается на свойство song компонента kenny. Подобное внедрение значения в свойство song компонента carl фактически равносильно операции присваивания, выполненной программно, как показано ниже:

```
Instrumentalist carl = new Instrumentalist();
carl.setSong(kenny.getSong());
```

Да! Наконец-то мы нашли более интересное применение выражениям на языке SpEL. Это очень простое выражение, но я не вижу более простого способа реализовать то же самое без SpEL.

И это еще не все! Мы только начинаем.



Рис. 2.1. Обращение
к свойству другого компонента
с использованием языка выражений Spring

Возможность обращения к свойствам – не единственное, что можно делать с компонентами. Точно так же можно вызывать методы компонентов. Например, представьте, что имеется компонент `songSelector` с методом `selectSong()`, возвращающим музыкальное произведение для исполнения. В этом случае Карл мог бы исполнять произведения, предлагаемые компонентом `songSelector`:

```
<property name="song" value="#{songSelector.selectSong()}" />
```

Теперь предположим, что по некоторой причине Карл пожелал, чтобы название произведения передавалось ему в виде строки, со-

стоящей только из заглавных символов. Нет проблем... для этого достаточно вызвать метод `toUpperCase()` строкового значения:

```
<property name="song" value="#{songSelector.selectSong().toUpperCase()}"/>
```

Этот трюк будет срабатывать всегда... пока метод `selectSong()` не вернет значение `null`. Если в какой-то момент `selectSong()` вернет `null`, во время вычисления выражения возникнет исключение `NullPointerException`.

Чтобы избежать появления жуткого исключения `NullPointerException` в выражении, следует задействовать защищенный оператор доступа к свойству:

```
<property name="song" value="#{songSelector.selectSong()?.toUpperCase()}"/>
```

Здесь вместо одиночной точки `(.)` для вызова метода `toUpperCase()` используется оператор `?..`. Этот оператор сначала проверяет, не является ли пустым элементом слева, и только потом выполняет обращение к элементу справа. То есть если `selectSong()` вернет значение `null`, тогда выражение не будет даже пытаться вызвать `toUpperCase()`.

Создание выражений, манипулирующих другими компонентами, – отличное начало. Но что, если потребуется вызвать статический метод класса или получить значение константы? Для этого необходимо познакомиться с особенностями использования типов в языке SpEL.

Использование типов

Ключом к использованию методов класса и констант в языке SpEL является оператор `T()`. Например, чтобы обратиться к Java-классу `Math` в выражении на языке SpEL, необходимо использовать оператор `T()`, как показано ниже:

```
T(java.lang.Math)
```

Результатом оператора `T()`, представленного здесь, является объект типа `Class`, представляющий класс `java.lang.Math`. Этот результат можно было бы даже внедрить в свойство компонента типа `Class`. Но истинная ценность значения, возвращаемого оператором `T()`, заключается в возможности доступа к статическим методам и константам указанного класса.

Например, предположим, что в свойство компонента необходимо внедрить значение π . В этом случае достаточно просто обратиться к константе PI в классе Math, как показано ниже:

```
<property name="multiplier" value="#{T(java.lang.Math).PI}" />
```

Аналогично с помощью оператора T() можно вызывать статические методы. Например, ниже показано, как в свойство компонента внедрить случайное число (в диапазоне от 0 до 1):

```
<property name="randomNumber" value="#{T(java.lang.Math).random()}" />
```

В момент запуска приложения, когда дело дойдет до инициализации свойства randomNumber, фреймворк Spring вызовет метод Math.random() и присвоит полученное значение этому свойству. Это еще один пример операций, которые трудно было бы реализовать без применения выражений на языке SpEL.

Теперь, когда мы добавили в наш арсенал несколько основных приемов использования выражений на языке SpEL, сделаем еще шаг и познакомимся с операциями, которые можно выполнять с этими простыми выражениями.

2.3.2. Выполнение операций со значениями SpEL

Язык SpEL предлагает несколько операций, которые могут применяться к значениям в выражениях. Эти операции перечислены в табл. 2.5.

Таблица 2.5. Язык SpEL включает несколько операторов для манипулирования значениями в выражениях

Типы операторов	Операторы
Арифметические	+, -, *, /, %, ^
Операторы отношений	<, >, ==, <=, >=, lt, gt, eq, le, ge
Логические	and, or, not, !
Условные	? : (трехместный), ?: (Элвис)
Регулярные выражения	matches

Сначала мы познакомимся с группой операторов, позволяющих выполнять в выражениях на языке SpEL основные математические операции со значениями.

Выполнение математических операций в языке SpEL

Язык SpEL поддерживает все основные арифметические операторы, имеющиеся в языке Java, плюс оператор «крышки» (^), выполняющий возведение в степень.

Например, чтобы сложить два числа, можно воспользоваться оператором +, как показано ниже:

```
<property name="adjustedAmount" value="#{counter.total + 42}" />
```

Здесь к значению свойства total компонента counter добавляется число 42. Обратите внимание, что хотя оба операнда в этом выражении являются числовыми, они не обязательно должны быть литералами. В данном случае левым операндом является полноценное выражение на языке SpEL.

Другие арифметические операторы языка SpEL действуют точно так же, как в языке Java. Оператор -, например, выполняет вычитание:

```
<property name="adjustedAmount" value="#{counter.total - 20}" />
```

Оператор * – умножение:

```
<property name="circumference"
  value="#{2 * T(java.lang.Math).PI * circle.radius}" />
```

Оператор / – деление:

```
<property name="average" value="#{counter.total / counter.count}" />
```

И оператор % – деление по модулю:

```
<property name="remainder" value="#{counter.total % counter.count}" />
```

В отличие от Java, язык SpEL также предлагает оператор возведения в степень:

```
<property name="area" value="#{T(java.lang.Math).PI * circle.radius ^ 2}" />
```

Даже при том, что речь идет об арифметических операторах языка SpEL, стоит упомянуть, что оператор + перегружен и способен выполнять конкатенацию строковых значений. Например:

```
<property name="fullName"
    value="#{performer.firstName + ' ' + performer.lastName}" />
```

Подобно тому как оператор `+` можно использовать для конкатенации строк в языке Java.

Сравнение значений

Часто бывает желательно сравнить два значения, чтобы определить, равны ли они или какое-то из них больше другого. Для подобного рода сравнений в языке SpEL имеются все необходимые операторы, которые присутствуют и в языке Java.

Например, чтобы выяснить равенство двух чисел, можно воспользоваться оператором равенства (`==`):

```
<property name="equal" value="#{counter.total == 100}" />
```

В данном случае, если предположить, что свойство `equal` имеет логический тип, ему будет присвоено значение `true`, если свойство `total` равно 100.

Аналогично для сравнения разных значений можно использовать операторы «меньше чем» (`<`) и «больше чем» (`>`). Кроме того, язык SpEL поддерживает операторы «больше или равно» (`>=`) и «меньше или равно» (`<=`). Например, ниже представлено вполне допустимое выражение на языке SpEL:

```
counter.total <= 100000
```

К сожалению, операторы «меньше чем» и «больше чем» могут вызывать проблемы при использовании в конфигурационных XML-файлах, так как в языке разметки XML они имеют специальное значение. Поэтому при использовании выражений на языке SpEL в XML-файлах¹ лучше использовать текстовые эквиваленты этих операторов. Например:

```
<property name="hasCapacity" value="#{counter.total le 100000}" />
```

Здесь оператор `le` означает «меньше или равно». Другие текстовые операторы сравнения перечислены в табл. 2.6.

¹ Как использовать выражения на языке SpEL за пределами конфигурационных XML-файлов, будет показано в главе 4.



Таблица 2.6. Язык SpEL включает несколько текстовых операторов для сравнения значений в выражениях

Операция	Символический оператор	Текстовый оператор
Равно	<code>==</code>	<code>eq</code>
Меньше чем	<code><</code>	<code>lt</code>
Меньше или равно	<code><=</code>	<code>le</code>
Больше чем	<code>></code>	<code>gt</code>
Больше или равно	<code>>=</code>	<code>ge</code>

Вы могли заметить, что даже при том, что символический оператор определения равенства (`==`) не вызывает проблем в XML, язык SpEL все равно предлагает альтернативный текстовый оператор `eq`, чтобы обеспечить непротиворечивость с другими операторами и потому что некоторые разработчики предпочитают использовать текстовые версии операторов.

Логические выражения

Это здорово, что имеется возможность выполнять сравнение в выражениях на языке SpEL, но что, если потребуется выполнить вычисления, опираясь на две операции сравнения? Или, например, потребуется инвертировать некоторое логическое значение? В этом вам помогут логические операторы. В табл. 2.7 перечислены все логические операторы, поддерживаемые языком SpEL.

Таблица 2.7. Язык SpEL включает несколько операторов для манипулирования логическими значениями в выражениях

Оператор	Операция
<code>and</code>	Логическая операция И; чтобы результатом выражения было значение <code>true</code> , оба операнда должны иметь значение <code>true</code>
<code>or</code>	Логическая операция ИЛИ; чтобы результатом выражения было значение <code>true</code> , хотя бы один из операндов должен иметь значение <code>true</code>
<code>not</code> или <code>!</code>	Логическая операция НЕ; инвертирует значение целевого выражения

Например, взгляните, как используется оператор `and`:

```
<property name="largeCircle"
  value="#{shape.kind == 'circle' and shape.perimeter gt 10000}"/>
```

В данном случае, если свойство `kind` компонента `shape` имеет значение `"circle"` и свойство `perimeter` имеет значение больше 10 000,

свойству `largeCircle` будет присвоено значение `true`. В противном случае оно получит значение `false`.

Инвертировать значение логического выражения можно с помощью двух операторов: либо с помощью символического оператора `!`, либо с помощью текстового оператора `not`. Например, ниже демонстрируется использование оператора `!`:

```
<property name="outOfStock" value="#{!product.available}"/>
```

что эквивалентно использованию оператора `not`:

```
<property name="outOfStock" value="#{not product.available}"/>
```

Странно, что в языке SpEL отсутствуют символические эквиваленты операторов `and` и `or`.

Условные вычисления

Как быть, если в выражении SpEL потребуется получить одно значение при выполнении условия и другое – в противном случае? Например, предположим, что Карл (компонент типа `Instrumentalist`) желает исполнять песню «*Jingle Bells*» только на фортепиано, а другие – на саксофоне. В этом случае можно использовать трехместный (тернарный) оператор `(?:)`:

```
<property name="instrument"
  value="#{songSelector.selectSong()=='Jingle Bells'?piano:saxophone}"/>
```

Трехместный оператор в языке SpEL действует точно так же, как в языке Java. В данном случае, если будет выбрана песня «*Jingle Bells*», в свойство `instrument` будет внедрена ссылка на компонент `piano`. В противном случае в него будет внедрена ссылка на компонент с идентификатором `saxophone`.

Типичная область применения трехместного оператора – проверка значения на равенство `null` и внедрение значения по умолчанию, если равенство соблюдается. Например, предположим, что необходимо подготовить Карла к исполнению того же произведения, которое должен исполнять Кенни, если только для Кенни было указано какое-либо произведение. В противном случае Карл должен исполнять произведение по умолчанию «*Greensleeves*». В такой ситуации можно было бы использовать трехместный оператор, как показано ниже:

```
<property name="song"
  value="#{kenny.song != null ? kenny.song : 'Greensleeves'}"/>
```

Это выражение построено правильно, но в нем имеется повторяющийся элемент – ссылка на свойство `kenny.song`. Язык SpEL предлагает разновидность трехместного оператора, упрощающую подобные выражения:

```
<property name="song" value="#{kenny.song ?: 'Greensleeves'}"/>
```

Как и в предыдущем примере, выражение вернет значение свойства `kenny.song` или «`Greensleeves`», если свойство `kenny.song` равно `null`. При таком способе использования оператор `?:` называют *оператором Элвиса*. Этим странным названием оператор обязан сравнению со смайликами – если повернуть его на 90 градусов, вопросительный знак будет напоминать прическу знаменитого Элвиса Пресли (Elvis Presley)¹.

Регулярные выражения

При работе с текстом иногда бывает желательно проверить его совпадение с некоторым шаблоном. Язык SpEL поддерживает сопоставление с шаблонами с помощью оператора `matches`.

Оператор `matches` пытается применить регулярное выражение (в аргументе справа) к строковому значению (в аргументе слева). Результатом выполнения оператора `matches` является логическое значение: `true` – если строка совпадает с шаблоном и `false` – в противном случае.

Для демонстрации оператора `matches` представим, что необходимо проверить в строке наличие допустимого адреса электронной почты. В этом случае можно применить оператор `matches`, как показано ниже:

```
<property name="validEmail" value=
  "#{admin.email matches '[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.\com'}"/>
```

Исследование тайн загадочного синтаксиса регулярных выражений выходит далеко за рамки этой книги. И я понимаю, что представленное здесь регулярное выражение не является достаточно на-

¹ Я тут ни при чем! Не я придумал это название. Но ведь действительно он напоминает прическу Элвиса?!

дежным, чтобы охватить все возможные случаи. Но для демонстрации применения оператора `matches` этого вполне достаточно.

Теперь, после знакомства с особенностями использования простых значений в выражениях, можно перейти к настоящей магии, которая в языке SpEL используется при работе с коллекциями.

2.3.3. Обработка коллекций на языке SpEL

Одни из самых необычных особенностей языка SpEL связаны с обработкой коллекций. Сослаться на отдельный элемент коллекции в языке SpEL можно точно так же, как в языке Java. Но в SpEL имеется мощный механизм выборки элементов коллекций на основе значений их свойств. Он также позволяет извлекать значения свойств элементов коллекций и составлять из них новые коллекции.

Для примера предположим, что имеется некоторый класс `City`, представленный ниже (для экономии места из объявления были исключены методы доступа):

```
package com.habuma.spel.cities;
public class City {
    private String name;
    private String state;
    private int population;
}
```

И что требуется сконфигурировать список объектов `City` с помощью элемента `<util:list>`, как показано ниже.

```
<util:list id="cities">
    <bean class="com.habuma.spel.cities.City"
        p:name="Chicago" p:state="IL" p:population="2853114"/>
    <bean class="com.habuma.spel.cities.City"
        p:name="Atlanta" p:state="GA" p:population="537958"/>
    <bean class="com.habuma.spel.cities.City"
        p:name="Dallas" p:state="TX" p:population="1279910"/>
    <bean class="com.habuma.spel.cities.City"
        p:name="Houston" p:state="TX" p:population="2242193"/>
    <bean class="com.habuma.spel.cities.City"
        p:name="Odessa" p:state="TX" p:population="90943"/>
    <bean class="com.habuma.spel.cities.City"
        p:name="El Paso" p:state="TX" p:population="613190"/>
    <bean class="com.habuma.spel.cities.City"
        p:name="Jal" p:state="NM" p:population="1996"/>
```



```
<bean class="com.habuma.spel.cities.City"
      p:name="Las Cruces" p:state="NM" p:population="91865"/>
</util:list>
```

Элемент `<util:list>` определен в пространстве имен `util` фреймворка Spring. Фактически он создает компонент типа `java.util.List`, содержащий все значения или компоненты, перечисленные в нем. В данном случае это список из восьми компонентов `City`.

Язык SpEL имеет несколько удобных операторов для работы с коллекциями, подобными этой.

Доступ к элементам коллекции

Чаще всего бывает необходимо извлечь из списка единственный элемент и внедрить его в свойство:

```
<property name="chosenCity" value="#{cities[2]}"/>
```

В данном случае я отобразил третий элемент из списка `cities` (отсчет элементов начинается с нуля) и внедрил его в свойство `chosenCity`. Для остроты ощущений можно также попробовать выбрать элемент списка случайным образом:

```
<property name="chosenCity"
          value="#{cities[T(java.lang.Math).random() * cities.size()]}"/>
```

В любом случае для доступа к элементу коллекции по его индексу должны использоваться квадратные скобки `([])`.

Оператор `[]` также можно использовать для извлечения элементов из отображений `java.util.Map`. Например, представим, что объекты `City` находятся в отображении (`Map`), где роль ключей играют названия городов. В этом случае элемент с городом `Dallas` (Даллас) можно извлечь так:

```
<property name="chosenCity" value="#{cities['Dallas']}"/>
```

Другой пример применения оператора `[]` – извлечение значения из коллекции типа `java.util.Properties`. Например, предположим, что в Spring необходимо загрузить файл с настройками свойств, используя элемент `<util:properties>`, как показано ниже:

```
<util:properties id="settings"
                 location="classpath:settings.properties"/>
```

Здесь компонент `settings` будет иметь тип `java.util.Properties` и содержать все элементы, имеющиеся в файле, с именами `settings.properties`. Благодаря SpEL можно получить доступ к свойствам в этом файле, как если бы они были элементами отображения типа Map. Например, ниже показано, как с помощью SpEL можно прочитать свойство `twitter.accessToken` из компонента `settings`:

```
<property name="accessToken" value="#{settings['twitter.accessToken']}
```

Помимо чтения свойств из коллекции `<util:properties>`, фреймворк Spring обеспечивает доступ в SpEL к двум специальным свойствам: `systemEnvironment` и `systemProperties`.

Свойство `systemEnvironment` содержит все переменные окружения системы, в которой выполняется приложение. Это обычная коллекция типа `java.util.Properties`, поэтому для доступа к ее элементам по ключам можно использовать квадратные скобки. Например, в моей системе MacOS X я могу внедрить путь к домашнему каталогу пользователя в свойство компонента, как показано ниже:

```
<property name="homePath" value="#{systemEnvironment['HOME']}
```

Свойство `systemProperties`, в свою очередь, содержит все параметры, которые были установлены при запуске приложения (обычно с помощью ключа `-D`). То есть, если виртуальная машина JVM была запущена с параметром `-Dapplication.home=/etc/myapp`, это значение можно внедрить в свойство `homePath` с помощью следующего выражения на языке SpEL:

```
<property name="homePath" value="#{systemProperties['application.home']}
```

Стоит также отметить, хотя это и не имеет прямого отношения к работе с коллекциями, что оператор `[]` можно применять к строковым значениям для извлечения одиночных символов. Например, следующее выражение вернет `"s"`:

```
'This is a test'[3]
```

Возможность доступа к отдельным элементам коллекций сама по себе удобна. Но в выражениях на языке SpEL имеется также возможность отбирать элементы коллекций, соответствующие определенным критериям. Попробуем сделать такую выборку из коллекции.



Выборка элементов коллекций

Допустим, что необходимо сократить список городов, оставив в нем только города с населением больше 100 000 человек. Один из способов заключается в том, чтобы внедрить весь компонент cities в свойство другого компонента и возложить на него все бремя отсеивания маленьких городов. Однако язык выражений SpEL предлагает более простое решение – оператор выборки (`.?[]`), как показано ниже:

```
<property name="bigCities" value="#{cities.? [population gt 100000]}"/>
```

Оператор выборки создаст новую коллекцию, которая будет включать элементы оригинальной коллекции, соответствующие критерию выбора, заключенному в квадратные скобки. В данном случае в свойство bigCities будет внедрен список городов (объектов City) с населением больше 100 000 человек.

В языке SpEL имеются также другие операторы выборки, `.^[]` и `.[$[]]`, позволяющие получить первый и последний (соответственно) элементы в выборке из коллекции. Например, чтобы получить первый крупный город из коллекции cities:

```
<property name="aBigCity" value="#{cities.^ [population gt 100000]}"/>
```

В процессе выборки объекты никак не упорядочиваются, поэтому в свойство aBigCity будет внедрен объект City, представляющий город Чикаго (Chicago). Аналогично объект City, представляющий город Эль-Пасо (El Paso), можно выбрать следующим образом:

```
<property name="aBigCity" value="#{cities.$ [population gt 100000]}"/>
```

Чуть ниже мы вернемся к проблеме выборки элементов из коллекций. Но сначала посмотрим, как отображаются свойства из оригинальной коллекции в новую коллекцию.

Отображение коллекций

Отображение коллекций связано с выбором определенного свойства каждого элемента оригинальной коллекции и помещением его в новую коллекцию. Оператор отображения (`.![]`) в языке SpEL выполняет именно эту операцию.

Например, предположим, что на основе списка объектов City необходимо получить список строк с именами городов. Чтобы полу-

чить такой список, можно выполнить внедрение в свойство cityNames, как показано ниже:

```
<property name="cityNames" value="#{cities.! [name]}"/>
```

В результате выполнения этого выражения свойству cityNames будет присвоен список объектов String с такими значениями, как Chicago, Atlanta, Dallas и т. д. Свойство name в квадратных скобках определяет, какие элементы будет содержать получившийся в результате список.

Но возможности отображения не ограничиваются отображением одиночных свойств. С небольшими дополнениями к предыдущему примеру можно получить список городов и названий штатов:

```
<property name="cityNames" value="#{cities.! [name + ', ' + state]}"/>
```

Теперь свойство cityNames будет содержать список значений, таких как «Chicago, IL», «Atlanta, GA» и «Dallas, TX».

И в последнем примере попробуем объединить операции создания выборки из коллекции и отображения. Ниже показано, как можно внедрить в свойство cityNames список имен только крупных городов:

```
<property name="cityNames"
  value="#{cities.? [population gt 100000].! [name + ', ' + state]}"/>
```

Поскольку результатом операции выборки является новый список объектов City, ничто не мешает отобразить его в новую коллекцию, содержащую имена всех крупных городов.

Этот пример демонстрирует возможность конструирования интересных (и сложных) выражений на языке SpEL из более простых подвыражений. Нетрудно понять, насколько широкие перспективы открывает эта возможность. Но не нужно обладать развитым воображением, чтобы понять, что она также еще и опасна. Выражения на языке SpEL – это всего лишь строки, которые сложно тестировать и для которых не поддерживается подсветка синтаксиса в интегрированных средах разработки.

Я рекомендую использовать выражения на языке SpEL везде, где они могут упростить внедрение, которое сложно (если вообще возможно) реализовать иными способами. Но будьте внимательны, не переусердствуйте. Не поддавайтесь искушению перенести максимально возможный объем логики в выражения на языке SpEL.



Мы еще вернемся к выражениям на языке SpEL позднее и рассмотрим возможность их применения в ситуациях, не связанных с конфигурированием компонентов. В главе 4 я вырву выражения SpEL из XML-файлов и задействую их в операциях внедрения через аннотации. Кроме того, в главе 10 будет показано, насколько важную роль играют выражения на языке SpEL в последней версии фреймворка Spring Security.

2.4. В заключение

В центре фреймворка Spring Framework находится контейнер Spring. В состав Spring входят несколько реализаций контейнера, но все они подразделяются на две категории. BeanFactory – самый простой вид контейнера, обеспечивает основу DI и службы связывания компонентов. Но когда возникает потребность в контейнере с расширенными возможностями, используется контейнер ApplicationContext.

В этой главе вы увидели, как связывать компоненты друг с другом внутри контейнера Spring. Обычно связывание внутри контейнера Spring выполняется с использованием XML-файла. Этот XML-файл содержит конфигурационную информацию для всех компонентов приложения, а также информацию, которая помогает контейнеру выполнять внедрение зависимостей, связывая компоненты друг с другом.

Теперь, когда вы знаете, как реализовать связывание компонентов внутри XML-файлов, я покажу, как организовать связывание с минимальным привлечением кода разметки XML. В главе 4 мы познакомимся с преимуществами автоматического связывания и аннотациями, позволяющими уменьшить объем конфигурационных XML-файлов в приложениях на основе фреймворка Spring.



Глава 3. Дополнительные способы связывания компонентов

В этой главе рассматриваются следующие темы:

- ❑ создание компонентов родитель/потомок;
- ❑ пользовательские редакторы свойств;
- ❑ постобработка компонентов;
- ❑ динамически управляемые компоненты.

Большинство людей имеют, по крайней мере, один ящик (а иногда даже целый шкаф или кабинет) в своем доме, где хранятся разные мелочи. Часто его называют просто «выдвижной ящик», но там может оказаться довольно много разных весьма полезных штуковин. В подобном месте всегда найдутся такие вещи, как рулетка, зажимы, ручки, карандаши, кнопки, несколько запасных батарей и т. д. Обычно эти предметы не используются каждый день, но вы точно знаете, что если отключат электричество, вы пороетесь в этом ящике и обязательно найдете батареи, которые вставите в ваш фонарь.

В главе 2 были представлены основные приемы связывания компонентов Spring, применяемые ежедневно. Несомненно, у вас будет масса возможностей использовать их в своих приложениях, основанных на Spring.

Однако в данной главе мы немного пороемся в выдвижном ящике Spring контейнеров. Хотя темы, охватываемые здесь, вполне практически для удовлетворения потребностей, они почти не найдут такого же применения, как те, что обсуждались в главе 2. Не часто будет необходимо заменить метод в компоненте или создать компонент, который знает свое имя. И не в каждом проекте потребуется вводить класс Ruby в Java-приложение, основанное на Spring.

Так как эта глава касается довольно необычных возможностей Spring, вы можете просто пропустить ее и перейти к знакомству с аспектно-ориентированными функциями Spring в главе 4. Но эта глава всегда будет к вашим услугам, ожидая, когда в ней возникнет потребность. И если вы окажетесь тут, то увидите, как можно свя-



зывать компоненты разными экстремальными способами. А начнем мы ее с того, как создать и расширить абстрактные компоненты.

3.1. Объявление родителей и потомков компонентов

Одной из основ объектно-ориентированного программирования является возможность создания класса, расширяющего другой класс. Новый класс наследует все свойства и методы родительского класса, но он в состоянии добавить новые свойства и методы и даже переопределить свойства и методы родителя. Если вы читаете эту книгу, то наверняка имеете опыт создания программ на языке Java и подклассы для вас отнюдь не новость. Но знаете ли вы, что компоненты в фреймворке Spring также могут быть «подкомпонентами» других компонентов?

Объявление `<bean>` в Spring обычно содержит атрибут `class`, чтобы указать тип компонента, и ноль или более элементов `<property>`, внедряющих свои значения в свойства компонента. Абсолютно все, что касается компонента, объявляется в одном месте: в нем самом, в его объявлении `<bean>`.

Но создание множества отдельных объявлений компонентов может сделать конфигурацию Spring громоздкой и хрупкой. Иерархия классов в Java создается, чтобы собрать общую функциональность и свойства в родительских классах, которые могут быть затем унаследованы дочерними классами. Именно по тем же причинам бывает удобно создавать компоненты, расширяющие и наследующие определения других компонентов. И это отличный способ сократить объем XML-файлов за счет устранения избыточных определений контекста Spring.

Для обеспечения «наследования» определений компонентов в элементе `<bean>` могут указываться два специальных атрибута:

- ❑ `parent` – определяет идентификатор компонента, который будет предком компонента с атрибутом `parent`. Атрибут `parent` указывает, что компонент расширяет Java-класс;
- ❑ `abstract` – если имеет значение `true`, указывает, что компонент объявлен как абстрактный. То есть экземпляр такого компонента никогда не должен создаваться фреймворком Spring.

Чтобы показать возможности подкомпонентов, вернемся к курсу «Spring Idol».

3.1.1. Абстрактные компоненты

Как рассказывалось в главе 2, Кенни был соперником тех, кто выступал в соревновании как музыкант (`Instrumentalist`). В частности, специализацией Кенни является игра на саксофоне. Кенни был объявлен в Spring как компонент следующим образом:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

Пока вы читали главу 2, в состязания вступил новый участник. Так совпало, что Давид – тоже саксофонист. Более того, он должен играть то же самое произведение, что и Кенни. Компонент Давида объявлен в Spring следующим образом:

```
<bean id="david"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

Сейчас у нас имеются два компонента, объявленные в Spring фактически одинаково. Как показано на рис. 3.1, единственное различие между двумя этими компонентами составляют их идентификаторы. Сейчас это может показаться небольшой проблемой, но представьте, что может произойти, если в конкурсе будут участвовать более 50 саксофонистов, которые все как один захотят исполнить одну и ту же мелодию «*Jingle Bells*».



Рис. 3.1. Два компонента одного типа и с одинаковыми свойствами, имеющими одинаковые значения

Очевидным решением проблемы может быть запрет нескольким участникам участвовать в конкурсе. Но мы уже создали прецедент, разрешив выступление Давида. Кроме того, нам нужен пример создания подкомпонента. Так что отложим пока открытие конкурса.

Другое решение – создать компонент, который будет родителем для всех соперничающих саксофонистов. Компонент `baseSaxophonist` должен будет исполнить следующий трюк:

```
<bean id="baseSaxophonist"
      class="com.springinaction.springidol.Instrumentalist"
      abstract="true">
    <property name="instrument" ref="saxophone" />
    <property name="song" value="Jingle Bells" />
</bean>
```

На первый взгляд компонент `baseSaxophonist` почти не отличается от компонентов `kenny` и `david`. Но обратите внимание на его атрибут `abstract` со значением `true`. Он сообщает фреймворку Spring не пытаться создавать экземпляра этого компонента... даже если от контейнера последует явный запрос. Во многом это тот же абстрактный Java-класс, экземпляр которого не может быть создан.

Хотя экземпляр компонента `baseSaxophonist` нельзя создать, он все еще очень полезен, потому что содержит общие свойства, присущие Кенни и Давиду. Поэтому сейчас мы можем объявить компоненты `kenny` и `david`, как показано ниже:

```
<bean id="kenny" parent="baseSaxophonist" />
<bean id="david" parent="baseSaxophonist" />
```

Атрибут `parent` свидетельствует, что оба компонента, `kenny` и `david`, будут наследовать определение от компонента `baseSaxophonist`. Обратите внимание на отсутствие атрибута `class` – компоненты `kenny` и `david` наследуют класс родительского компонента, а также его свойства. Мы избавились от избыточного кода XML, и элементы `<bean>` стали проще, как показано на рис. 3.2.

Здесь уместно упомянуть, что родительские компоненты могут быть и неабстрактными. Разумеется, вполне возможно создать подкомпонент, расширяющий конкретный компонент. Но в данном примере известно, что у фреймворка Spring нет причин создавать экземпляр компонента `baseSaxophonist`, поэтому он был объявлен абстрактным.

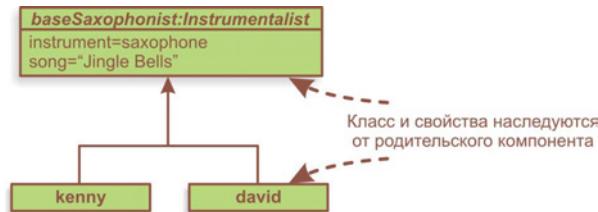


Рис. 3.2. Компоненты kenny и david

совместно используют общую конфигурацию.

Создавая родительский компонент с общими свойствами и наследуя его, можно исключить избыточную конфигурацию

Переопределение наследуемых свойств

Предположим, что на конкурсе появился еще один саксофонист (я говорил, что это будет происходить). Но вместо «Jingle Bells» он будет исполнять мелодию «Mary Had a Little Lamb». Означает ли это невозможность переопределения baseSaxophonist, когда будет объявляться новый участник?

Конечно нет. Мы все еще можем наследовать компонент baseSaxophonist. Но вместо того чтобы использовать значения всех наследуемых свойств, можно переопределить свойство song. Ниже следует объявление нового саксофониста:

```
<bean id="frank" parent="baseSaxophonist">
    <property name="song" value="Mary had a little lamb" />
</bean>
```

Компонент frank также наследует класс и свойства компонента baseSaxophonist. Но, как показано на рис. 3.3, он переопределяет свойство song так, что он может исполнять песню о девочке и ее макнатом друге.

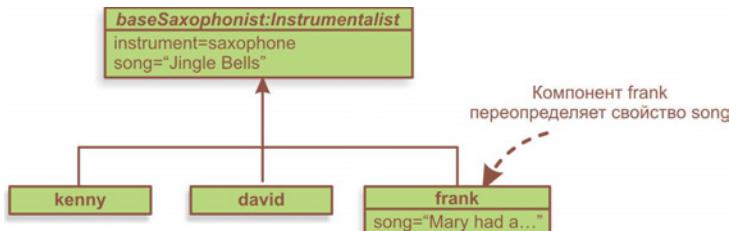


Рис. 3.3. Компонент frank наследует baseSaxophonistbean, но переопределяет свойство song

Во многих отношениях объявления родительских и дочерних компонентов отражают возможность определения родительских и дочерних классов в Java. Но затаите дыхание... я покажу, что наследование в Spring может то, чего невозможно сделать в Java.

3.1.2. Общие абстрактные свойства

В конкурсе талантов «Spring Idol» может быть несколько участников с музыкальными способностями. Как было показано выше, у нас есть несколько музыкантов, исполняющих мелодии на своих инструментах. Но в конкурсе могут также участвовать певцы.

Предположим, что в конкурсе талантов «Spring Idol» имеются два участника, певец и гитарист, исполняющие одну и ту же песню. При конфигурировании их в виде разных компонентов их объявления могут выглядеть, как показано ниже:

```
<bean id="taylor"
      class="com.springinaction.springidol.Vocalist">
    <property name="song" value="Somewhere Over the Rainbow" />
</bean>

<bean id="stevie"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="instrument" ref="guitar" />
    <property name="song" value="Somewhere Over the Rainbow" />
</bean>
```

Как и прежде, эти два компонента совместно используют некоторую общую конфигурацию. Определению, они оба имеют одно и то же значение атрибута `song`. Но не похожи на объявления в предыдущем разделе – эти два компонента относятся к разным классам. Поэтому для них нельзя создать общий родительский компонент. Или можно?

В языке Java классы-потомки наследуют общий базовый тип, который определен их родительским компонентом. Это означает, что в Java нет способа для класса потомка расширить общий тип и унаследовать при этом свойства и/или методы, но не унаследовать класс общего родителя. Однако в Spring подкомпонент не обязан наследовать тип общего родителя.

Два компонента с совершенно разными значениями атрибута `class` все еще могут иметь общие настройки свойств, унаследованные от родительского компонента.

Рассмотрим следующее объявление родительского абстрактного компонента:

```
<bean id="basePerformer" abstract="true">
    <property name="song" value="Somewhere Over the Rainbow" />
</bean>
```

Компонент `basePerformer` объявляет общее свойство `song`, которое унаследуют два наших исполнителя. Но обратите внимание на отсутствие в нем атрибута `class`. Это нормально, потому что каждый компонент-потомок будет определять свой тип в собственном атрибуте `class`. Ниже приводится новое определение компонентов `taylor` и `stevie`:

```
<bean id="taylor"
      class="com.springinaction.springidol.Vocalist"
      parent="basePerformer" />

<bean id="stevie"
      class="com.springinaction.springidol.Instrumentalist"
      parent="basePerformer">
    <property name="instrument" ref="guitar" />
</bean>
```

Обратите внимание, что наряду с другими атрибутами оба компонента используют атрибуты `class` и `parent`. Это обеспечивает возможность для двух и более отличных друг от друга компонентов наследовать свойства общего базового компонента. Как показано на рис. 3.4, значение свойства `song` наследуется из компонента `basePerformer`, даже если каждый наследник имеет совершенно иной, неродственный тип.

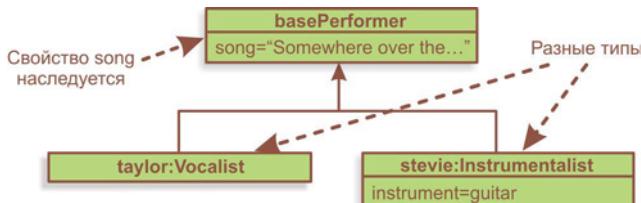


Рис. 3.4. Родительские компоненты не имеют определенного типа. Они могут использоваться только для хранения общей конфигурации, наследуемой компонентами других типов

Возможность создания подкомпонентов обычно используется для уменьшения объема кода XML, необходимого для объявления аспектов и транзакций. В главе 6 вы увидите, как применение подкомпонентов помогает значительно уменьшить количество избыточного XML-кода при объявлении транзакций `TransactionProxyFactoryBean`.

А пока продолжим копаться в ящике, чтобы посмотреть, какие еще полезные инструменты можно найти. Мы уже познакомились с приемом внедрения через конструкторы и методы доступа в главе 2. Теперь посмотрим, как Spring обеспечивает другие способы внедрения, позволяющие декларативно вводить методы в компоненты, фактически изменяя их функциональные возможности, без необходимости менять лежащий в их основе программный код на языке Java.

3.2. Внедрение методов

В главе 2 описывались две основные формы *внедрения зависимостей* (DI). Внедрение через конструктор позволяет настраивать компоненты, передавая значения через аргументы конструкторов. Аналогично внедрение через методы доступа позволяет настраивать компонент, передавая значения через аргументы методов записи. Прежде чем эта книга закончится, в ней будут представлены сотни примеров внедрения через методы доступа и, может быть, немногим меньше примеров внедрения через конструкторы.

Но в этом разделе я хотел бы показать необычную форму DI, получившую название «внедренных методов». При использовании приема внедрения через методы доступа или конструкторы производится внедрение значений в свойства компонента. Но, как показано на рис. 3.5, прием внедрения методов совершенно отличается от двух предыдущих, потому что позволяет внедрять в компоненты определения целых методов.



Рис. 3.5. Внедрение методов – это разновидность внедрения, когда методы класса заменяются альтернативными реализациями

Некоторые языки программирования, такие как Ruby, позволяют добавлять новые методы в любой класс непосредственно во время выполнения, без изменения определения класса. Например, если

в приложении на языке Ruby потребуется добавить новый метод в класс `String`, который будет выводить длину строки, достаточно будет лишь определить новый метод:

```
class String
  def print_length
    puts "This string is #{self.length} characters long"
  end
end
```

После добавления определения метода его можно вызвать для любого созданного в программе объекта `String`. Например, следующий фрагмент:

```
message = "Hello"
message.print_length
```

выведет «`This string is 5 characters long`» в стандартный поток вывода.

Но это Ruby. Язык Java не такой гибкий. Тем не менее фреймворк Spring предоставляет Java-программистам возможность внедрения методов в Java-классы во время выполнения. Пускай он не такой элегантный, как в языке Ruby, но все же это шаг в нужном направлении.

Фреймворк Spring поддерживает две формы внедрения методов:

- ❑ *замещение метода* – позволяет во время выполнения заменить существующий метод (абстрактный или конкретный) новой реализацией;
- ❑ *внедрение метода чтения* – позволяет во время выполнения заменить существующий метод (абстрактный или конкретный) новой реализацией, возвращающей определенный компонент из контекста Spring.

Для начала рассмотрим, как в Spring действует поддержка более универсального способа замещения методов.

3.2.1. Основы замещения методов

Вам нравятся шоу иллюзионистов? Фокусники используют ловкость рук и отвлечение внимания, чтобы прямо на наших глазах делать казалось бы невозможные вещи. Один из наших излюбленных трюков – когда фокусник помещает своего ассистента в ящик, кружит вокруг ящика, бубнит какие-то магические слова, потом... вуала! Ящик открывается, и в нем вместо ассистента оказывается тигр.

Так случилось, что Гарри, обещанный нами фокусник, только что включился в конкурс талантов «Spring Idol» и будет исполнять наш любимый фокус. Позвольте мне представить вам Гарри – первый из класса `Magician`, определение которого приводится в листинге 3.1.

Листинг 3.1. Фокусник и его черный ящик

```
package com.springinaction.springidol;

public class Magician implements Performer {
    public Magician() {}

    public void perform() throws PerformanceException {
        System.out.println(magicWords);
        System.out.println("The magic box contains... ");
        System.out.println(magicBox.getContents()); // Исследует содержимое
}                                              // ящика

// внедрение
private MagicBox magicBox;
public void setMagicBox(MagicBox magicBox) {
    this.magicBox = magicBox; // Внедрение волшебного ящика
}

private String magicWords;

public void setMagicWords(String magicWords) {
    this.magicWords = magicWords;
}
}
```

Как видите, класс `Magician` имеет два свойства, которые могут быть установлены с помощью Spring DI. Фокуснику нужны какие-нибудь магические слова, чтобы магия сработала, поэтому желательно установить значение свойства `magicWords`. Но что более важно, фокуснику нужно передать черный ящик через свойство `magicBox`. Коль скоро речь зашла о черном ящике, его реализацию можно увидеть в листинге 3.2.

Листинг 3.2. Реализация черного ящика с симпатичной ассистенткой внутри... или нет?

```
package com.springinaction.springidol;

public class MagicBoxImpl implements MagicBox {
```

```
public MagicBoxImpl() {}

public String getContents() {
    return "A beautiful assistant"; // Симпатичная ассистентка в ящике
}
}
```

Ключевым методом в классе MagicBoxImpl, на который следует обратить внимание, является метод getContents(). Заметьте, что он всегда возвращает строку «A beautiful assistant» (симпатичная ассистентка), но, как вскоре будет показано, в действительности все выглядит не так, как кажется. Однако, прежде чем я покажу фокус, взгляните, как Гарри и его черный ящик связаны с контекстом приложения:

```
<bean id="harry"
      class="com.springinaction.springidol.Magician">
    <property name="magicBox" ref="magicBox" />
    <property name="magicWords" value="Bippity boppity boo" />
</bean>

<bean id="magicBox"
      class="com.springinaction.springidol.MagicBoxImpl" />
```

Если вы еще не видели этот фокус прежде, знайте, что фокусник всегда дразнит аудиторию, закрывая ящик, а затем открывая его снова, чтобы показать, что ассистентка все еще там. Гарри действует точно так же. Начнем фокус с помощью следующего фрагмента:

```
ApplicationContext ctx = ... // загрузка контекста Spring
Performer magician = (Performer) ctx.getBean("harry");
magician.perform();
```

Когда этот фрагмент извлечет компонент harry из контекста приложения и вызовет метод perform(), вы увидете то, что ожидаете увидеть:

```
Bippity boppity boo
The magic box contains...
A beautiful assistant
```

Выход этих строк не должен стать сюрпризом для вас. В конце концов, метод getContents() объекта MagicBoxImpl всегда должен воз-

вращать строку «A beautiful assistant». Но, как я сказал, это лишь подразнивание. Гарри пока не выполняет настоящий фокус. Однако пора бы начать представление, так что давайте настроим XML-конфигурацию, чтобы она выглядела следующим образом:

```
<bean id="magicBox"
      class="com.springinaction.springidol.MagicBoxImpl">
    <replaced-method
        name="getContents"
        replacer="tigerReplacer" />
</bean>

<bean id="tigerReplacer"
      class="com.springinaction.springidol.TigerReplacer" />
```

Сейчас в компоненте `magicBox` имеется элемент `<replaced-method>` (см. рис. 3.6). Как следует из названия, этот элемент используется для замены метода новой реализацией. В данном случае атрибут `name` указывает имя замещаемого метода `getContents()`. А атрибут `replacer` ссылается на компонент `tigerReplacer`, реализующий замену.

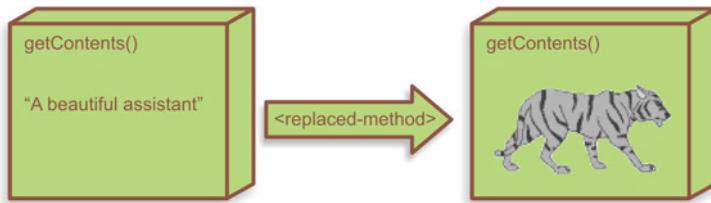


Рис. 3.6. Использование элемента `<replaced-method>` для внедрения облегчает замещение метода `getContents()`, возвращающего строку «A beautiful assistant», иной реализацией, которая создает тигра

Вот где настоящая ловкость рук. Компонент `tigerReplacer` – это объект класса `TigerReplacer`, который определен в листинге 3.3.

Листинг 3.3. TigerReplacer, который подменяет реализацию метода

```
package com.springinaction.springidol;
import java.lang.reflect.Method;
import org.springframework.beans.factory.support.MethodReplacer;

public class TigerReplacer implements MethodReplacer {
```

```
public Object reimplement(Object target, Method method, // Подмена
    Object[] args) throws Throwable { // метода

    return "A ferocious tiger"; // Помещает тигра в ящик
}

}
```

Класс `TigerReplacer` реализует интерфейс `MethodReplacer`, требующий реализации единственного метода `reimplement()`. Этот метод принимает три аргумента: объект, в котором будет производиться замещение метода, метод, подлежащий замещению, и массив аргументов, принимаемых методом. В нашем случае аргументы отсутствуют, но их можно передать при необходимости.

Тело метода `reimplement()` фактически становится новой реализацией метода `getContents()` черного ящика. В нашем примере единственное, что требуется, – это вернуть строку «*A ferocious tiger*» (свирепый тигр). Фактически прежнее содержимое ящика заменяется тигром, как показано на рис. 3.6.

Если теперь вызвать метод `perform()`, он выведет следующее:

```
Bippity boppity boo
The magic box contains...
A ferocious tiger
```

Та-да! Симпатичная ассистентка пропала, и вместо нее появился свирепый тигр, причем без изменения существующей реализации `MagicBoxImpl`. Фокус был успешно выполнен с помощью элемента `<replaced-method>`.

Стоит отметить, что хотя `MagicBoxImpl` имеет конкретную реализацию метода `getContents()`, метод `getContents()` точно так же можно было бы объявить абстрактным. В самом деле, прием внедрения метода с успехом может использоваться, когда фактическая реализация замещаемого метода не известна до момента развертывания. К этому моменту класс с замещающей реализацией метода можно оформить в виде JAR-файла и поместить его в библиотеку классов (`classpath`) приложения.

Замещение методов – это, конечно, изящный трюк. Но есть более специализированная форма внедрения методов, позволяющая сре-де выполнения связывать компоненты через метод чтения. Посмотрим, как выполняется внедрение через метод чтения в компонентах Spring.

3.2.2. Использование внедрения методов чтения

Внедрение через метод чтения – это особый случай внедрения методов, когда объявляется метод (обычно абстрактный), возвращающий компонент определенного типа, но фактически возвращаемый компонент определяется в контексте Spring.

Для иллюстрации рассмотрим новую форму класса Instrumentalist с внедряемым методом, как показано в листинге 3.4.

Листинг 3.4. Класс Instrumentalist с внедряемым методом

```
package com.springinaction.springidol;

public abstract class Instrumentalist implements Performer {
    public Instrumentalist() {}

    public void perform() throws PerformanceException {
        System.out.print("Playing " + song + " : ");
        getInstrument().play(); // Используется внедряемый метод
    } // getInstrument()

    private String song;

    public void setSong(String song) {
        this.song = song;
    }

    public abstract Instrument getInstrument(); // Внедряемый метод
}
```

В отличие от оригинального класса Instrumentalist, этот класс не получает инструмент посредством внедрения через метод записи. Вместо этого он имеет абстрактный метод getInstrument(), который будет возвращать инструмент исполнителя. Но если getInstrument() – абстрактный метод, тогда возникает вопрос, как этот метод получит реализацию.

Один из возможных подходов – использовать прием замещения методов, как описано в предыдущем разделе. Но для этого придется написать класс, реализующий интерфейс MethodReplacer, хотя нам требуется всего лишь переопределить метод getInstrument(), чтобы он возвращал определенный компонент.

Для поддержки внедрения методов чтения фреймворк Spring предлагает элемент `<lookup-method>`. Как и `<replaced-method>`, элемент `<lookup-method>` замещает существующий метод новой реализацией во время выполнения. Но элемент `<lookup-method>` – это упрощенный вариант элемента `<replaced-method>`, для которого нужно определить компонент в контексте Spring, возвращающий замещающий метод. Элемент `<lookup-method>` этого не требует. При его использовании не надо писать класс, реализующий интерфейс `MethodReplacer`.

Следующий фрагмент XML-кода демонстрирует использование элемента `<lookup-method>` для замещения метода `getInstrument()` другим методом, который вернет ссылку на компонент `guitar`.

```
<bean id="stevie"
      class="com.springinaction.springidol.Instrumentalist">
    <lookup-method name="getInstrument" bean="guitar" />
    <property name="song" value="Greensleeves" />
</bean>
```

Как и в элементе `<replaced-method>`, атрибут `name` элемента `<lookup-method>` определяет замещаемый метод. Здесь замещается метод `getInstrument()`. Атрибут `bean` определяет компонент, возвращаемый методом `getInstrument()`. В нашем случае это компонент с идентификатором `guitar`. В результате метод `getInstrument()` фактически будет замещен следующей реализацией:

```
public Instrument getInstrument() {
    ApplicationContext ctx = ...;

    return (Instrument) ctx.getBean("guitar");
}
```

Сам по себе прием внедрения методов чтения является всего лишь разновидностью внедрения через методы записи. Однако это имеет значение, только когда компонент имеет область действия `prototype`:

```
<bean id="guitar" class="com.springinaction.springidol.Guitar"
      scope="prototype" />
```

Даже если компонент имеет область видимости `prototype`, при использовании приема внедрения через метод записи компонент `guitar` будет внедрен в свойство только однажды. Использование приема



внедрения методов чтения гарантирует, что каждый вызов метода `getInstrument()` будет возвращать различные гитары. Это может пригодиться, если гитарист порвет струну во время выступления и потребует другой инструмент.

Важно помнить, что при использовании элемента `<lookup-method>` для внедрения метода чтения совсем необязательно, чтобы замещаемый метод был методом чтения (то есть методом, имя которого начинается со слова `get`). С помощью элемента `<lookup-method>` можно заменить любой метод, возвращающий некоторое значение.

Следует также отметить, что даже при том, что прием внедрения методов позволяет замещать их реализации, нет никакой возможности изменить сигнатуру метода. Типы параметров и возвращаемого значения должны оставаться такими, какие они есть. В случае с элементом `<lookup-method>` это означает, что атрибут `bean` должен ссылаться на компонент, тип которого совместим с типом значения, возвращаемого методом (`Instrument` в предыдущем примере).

3.3. Внедрение не-Spring компонентов

Как рассказывалось в главе 2, одна из главных задач фреймворка Spring – настраивать экземпляры компонентов. Но до сих пор всегда подразумевалось одно обстоятельство: контейнер Spring может настраивать только компоненты, экземпляры которых он создает сам. На первый взгляд может показаться странным, но это представляет некоторые проблемы. Не все объекты, имеющиеся в приложении, создаются контейнером Spring. Рассмотрим следующие возможные ситуации:

- ❑ Обычно *JSP-теги* реализуются веб-контейнером, в рамках которого выполняется приложение. Если JSP-тег нуждается во взаимодействии с какими-то другими объектами, он должен создавать их сам.
- ❑ *Доменные объекты*, которые типично создаются во время выполнения инструментами ORM (такими как Hibernate или iBATIS). В расширенной доменной модели доменные объекты обладают информацией о состоянии, и поведением. Но если нет возможности внедрить служебные объекты в доменные объекты, то доменные объекты должны создавать их сами или иметь собственную реализацию логики их поведения.

Существует еще одна веская причина, почему может потребоваться возложить на фреймворк Spring обязанность по настройке объ-

ектов, которых он не создает. Например, предположим, что мы явно создаем экземпляр класса `Instrumentalist` из примера «Spring Idol»:

```
Instrumentalist pianist = new Instrumentalist();
pianist.perform();
```

Так как `Instrumentalist` – это РОЮ, нет никаких причин, препятствующих явному созданию его экземпляра. Но когда вызывается метод `perform()`, может возникнуть исключение `NullPointerException`, потому что, несмотря на отсутствие препятствий непосредственного создания экземпляра класса `Instrumentalist`, его свойство `instrument` может иметь значение `null`.

Естественно, можно вручную настроить свойства объекта `Instrumentalist`. Например:

```
Piano piano = new Piano();
pianist.setInstrument(piano);
pianist.setSong("Chopsticks");
```

Даже при том, что вручную настроенные свойства будут работать, такой подход лишает преимущества отделения настроек от программного кода. Кроме того, если бы экземпляры класса `Instrumentalist` создавались механизмом ORM, мы бы не смогли изменять настройки их свойств.

К счастью, в версии Spring 2.0 появилась возможность декларативно настраивать компоненты, экземпляры которых создаются за пределами Spring. Суть в том, что Spring настраивает компоненты, но не создает их экземпляры.

Рассмотрим компонент `Instrumentalist`, явно созданный выше. В идеале хотелось бы сконфигурировать компонент `pianist` за пределами программного кода и позволить фреймворку Spring внедрить значения в его свойства `instrument` и `song`. Следующий фрагмент XML-кода демонстрирует, как этого добиться.

```
<bean id="pianist"
      class="com.springinaction.springidol.Instrumentalist"
      abstract="true">
    <property name="song" value="Chopsticks" />
    <property name="instrument">
      <bean class="com.springinaction.springidol.Piano" />
    </property>
</bean>
```

В таком объявлении компонента нет ничего необычного. Компонент `pianist` объявляется как объект класса `Instrumentalist`. А его свойства `song` и `instrument` связываются со своими значениями. Это самый заурядный компонент Spring, за исключением одной маленькой детали: его атрибут `abstract` имеет значение `true`.

Как рассказывалось выше, атрибут `abstract` со значением `true` сообщает фреймворку Spring, что для этого компонента не требуется создавать экземпляр класса. Этот прием часто используется при объявлении родительских компонентов, которые будут расширяться дочерними компонентами. Но в данном случае мы просто указываем фреймворку Spring, что компонент `pianist` не должен создаваться им, – это будет сделано без участия фреймворка.

На самом деле компонент `pianist` служит для Spring лишь шаблоном настройки экземпляров `Instrumentalist`, созданных за его пределами. После определения шаблона необходим некоторый способ связывания его с классом `Instrumentalist`. Для этого нужно аннотировать класс `Instrumentalist` аннотацией `@Configurable`:

```
package com.springinaction.springidol;
import org.springframework.beans.factory.annotation.Configurable;

@Configuration("pianist")
public class Instrumentalist implements Performer {
    ...
}
```

Аннотация `@Configurable` играет двоякую роль:

- ❑ во-первых, она показывает, что экземпляр класса `Instrumentalist` может быть сконфигурирован фреймворком Spring, даже при создании за его пределами;
- ❑ она также связывает класс `Instrumentalist` и компонент с идентификатором `pianist`. При настройке экземпляра класса `Instrumentalist` фреймворк Spring будет использовать определение компонента `pianist` как шаблон.

Но как Spring узнает, как настраивать компоненты с аннотацией `@Configurable`? За это отвечает следующий элемент в конфигурации Spring:

```
<aop:spring-configured />
```

Элемент `<aop:spring-configured>` – это один из множества новых элементов, появившихся в Spring 2.0. Они подсказывают фреймвор-

ку Spring, что эти компоненты необходимо конфигурировать, даже если они созданы где-то в другом месте.

За кулисами элемент <aop:spring-configured> устанавливает аспект AspectJ с точкой внедрения, которая срабатывает при создании любого компонента с аннотацией @Configurable. Сразу после создания компонента в игру вступает аспект и внедряет свойства в новый экземпляр, опираясь на шаблон <bean> в конфигурации Spring.

Так как аспект, сконфигурированный фреймворком Spring, является аспектом AspectJ, приложение должно запускаться под управлением JVM с включенной поддержкой AspectJ. Проще всего включить эту поддержку в Java 5 JVM – это запустить ее со следующим аргументом JVM:

```
-javaagent:/path/to/aspectjweaver.jar
```

Фактически этот аргумент предписывает виртуальной машине JVM выполнить внедрение любых аспектов AspectJ на этапе загрузки. Для этого она должна знать, где находятся классы AspectJ. Вам потребуется заменить

```
/path/to/aspectjweaver.jar
```

действительным путем к файлу aspectjweaver.jar в вашей системе (если, конечно, он не находится в каталоге /path/to).

Чтобы продемонстрировать способность фреймворка Spring конфигурировать компоненты, созданные за его пределами, в этом разделе явно был создан экземпляр Instrumentalist. Тем не менее, как уже упоминалось выше, в действующих приложениях подобные конфигурируемые компоненты, вероятнее всего, будут создаваться механизмом ORM или некоторой сторонней библиотекой.

А теперь рассмотрим, как редактор свойств фреймворка Spring выполняет простую работу по внедрению сложных значений, опираясь на строковое представление.

3.4. Пользовательские редакторы свойств

По мере изучения этой книги можно заметить несколько примеров, где сложные свойства устанавливаются посредством простого строкового значения. Примером может служить связывание веб-служб

с использованием фабричного компонента JaxRpcPortProxyFactoryBean. Одно из свойств JaxRpcPortProxyFactoryBean, которое нужно установить, – это wsdlDocumentUrl. Это свойство имеет тип java.net.URL. Но вместо того чтобы создавать компонент java.net.URL и внедрять его в это свойство, можно сконфигурировать его, используя строку, как показано ниже:

```
<property name="wsdlDocumentUrl"
  value="http://www.xmethods.net/sd/BabelFishService.wsdl" />
```

За кулисами фреймворк Spring преобразует строковое значение в объект URL. В действительности суть этого трюка не в каких-то особых возможностях Spring, а скорее в малоизвестной возможности оригинального JavaBeans API. Интерфейс java.beans.PropertyEditor позволяет определять, как значения типа String должны отображаться в значения других типов. Удобная реализация этого интерфейса, java.beans.PropertyEditorSupport, имеет два метода, интересующие нас:

- ❑ getAsText() возвращает строковое представление значения свойства;
- ❑ setAsText(String value) присваивает свойству компонента значение, переданное в виде строки.

При попытке присвоить нестроковому свойству значение типа String вызывается метод setAsText() для выполнения преобразования. Точно так же, когда требуется получить строковое представление значения свойства, вызывается метод getAsText().

В состав Spring входит несколько собственных редакторов, основанных на PropertyEditorSupport, включая org.springframework.beans.propertyeditors.URLEditor, который преобразует строковые значения в объекты java.net.URL и обратно. В табл. 3.1 приводится выборочный список редакторов, входящих в состав Spring.

Таблица 3.1. В состав Spring входят несколько собственных редакторов свойств, которые автоматически преобразуют внедряемые строковые значения в более сложные типы

Редактор	Описание
ClassEditor	Устанавливает свойство типа java.lang.Class на основе строки, содержащей полное имя класса
CustomDateEditor	Устанавливает свойство типа java.util.Date на основе строки и с использованием пользовательского объекта java.text.DateFormat

Таблица 3.1 (окончание)

Редактор	Описание
FileEditor	Устанавливает свойство типа <code>java.io.File</code> на основе строки, содержащей полный путь к файлу
LocaleEditor	Устанавливает свойство типа <code>java.util.Locale</code> на основе строки, содержащей текстовое представление локали (например, <code>en_US</code>)
StringArrayPropertyEditor	Преобразует строку с подстроками, разделенными запятыми, в массив строк
StringTrimmerEditor	Автоматически отсекает пробелы у строковых свойств. Пустые строки могут быть преобразованы в значение <code>null</code>
URLEditor	Устанавливает свойство типа <code>java.net.URL</code> на основе строки, содержащей адрес URL

В дополнение к обычным редакторам, перечисленным в табл. 3.1, можно написать собственный редактор, расширяя класс `PropertyEditorSupport`. Например, допустим, что в приложении имеется компонент `Contact`, который удобно использовать для хранения информации о сотрудниках в организации. Кроме всего прочего, объект `Contact` имеет свойство `phoneNumber`, хранящее номер телефона:

```
public Contact {
    private PhoneNumber phoneNumber;

    public void setPhoneNumber(PhoneNumber phoneNumber) {
        this.phoneNumber = phoneNumber;
    }
}
```

Свойство `phoneNumber` имеет тип `PhoneNumber` и определяется следующим образом:

```
public PhoneNumber {
    private String areaCode;
    private String prefix;
    private String number;

    public PhoneNumber() { }

    public PhoneNumber(String areaCode, String prefix,
                      String number) {
        this.areaCode = areaCode;
```

```
    this.prefix = prefix;
    this.number = number;
}
...
}
```

Используя базовый прием связывания, описанный в главе 2, связать объект `PhoneNumber` со свойством `phoneNumber` в компоненте `Contact` можно следующим образом:

```
<beans>
    <bean id="infoPhone"
          class="com.springinaction.chapter03.propeditor.PhoneNumber">
        <constructor-arg value="888" />
        <constructor-arg value="555" />
        <constructor-arg value="1212" />
    </bean>
    <bean id="contact"
          class="com.springinaction.chapter03.propeditor.Contact">
        <property name="phoneNumber" ref="infoPhone" />
    </bean>
</beans>
```

Обратите внимание, что требуется определить отдельный компонент `infoPhone`, настроить объект `PhoneNumber` и затем связать свойство `phoneNumber` с определением компонента.

Допустим, что вместо этого вы написали свой редактор `PhoneEditor`, как показано ниже:

```
public class PhoneEditor
    extends java.beans.PropertyEditorSupport {

    public void setAsText(String textView) {
        String stripped = stripNonNumeric(textValue);

        String areaCode = stripped.substring(0,3);
        String prefix = stripped.substring(3,6);
        String number = stripped.substring(6);
        PhoneNumber phone = new PhoneNumber(areaCode, prefix, number);
        setValue(phone);
    }

    private String stripNonNumeric(String original) {
```

```

        StringBuffer allNumeric = new StringBuffer();

        for(int i=0; i<original.length(); i++) {
            char c = original.charAt(i);
            if(Character.isDigit(c)) {
                allNumeric.append(c);
            }
        }
        return allNumeric.toString();
    }
}

```

Теперь единственное, что осталось сделать, – заставить фреймворк Spring использовать пользовательский редактор свойств при связывании свойств компонентов. Для этого следует использовать CustomEditorConfigurer. CustomEditorConfigurer – это BeanFactoryPostProcessor, который загружает пользовательский редактор в BeanFactory, вызывая метод registerCustomEditor(). (При желании можно вызвать метод registerCustomEditor() в своем программном коде, после создания экземпляра фабрики компонентов).

Следующий фрагмент XML-кода в файле конфигурации компонента подскажет фреймворку Spring о необходимости зарегистрировать PhoneEditor в качестве пользовательского редактора:

```

<bean
    class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="com.springinaction.chapter03.propeditor.PhoneNumber">
                <bean id="phoneEditor"
                    class="com.springinaction.chapter03.propeditor.PhoneEditor">
                </bean>
            </entry>
        </map>
    </property>
</bean>

```

Теперь вы сможете сконфигурировать свойство phoneNumber объекта Contact, используя простое строковое значение и не создавая отдельный компонент infoPhone:

```

<bean id="contact"
    class="com.springinaction.chapter03.propeditor.Contact">
    <property name="phoneNumber" value="888-555-1212" />
</bean>

```

Обратите внимание, что многие редакторы, входящие в состав фреймворка Spring (такие как `URLEditor` и `LocaleEditor`), к моменту запуска контейнера уже будут зарегистрированы в фабрике компонентов. Их не требуется регистрировать явно, используя `CustomEditorConfigurer`.

Редакторы свойств – это лишь один из способов повлиять на порядок создания и внедрения компонентов в Spring. Существуют и другие типы компонентов, выполняющие в контейнере Spring особые функции. В следующем разделе будет показано, как создать несколько специальных компонентов, позволяющих управлять процессом связывания компонентов в контейнере Spring.

3.5. Специальные компоненты Spring

Большинство компонентов, определяемых в контейнере Spring, обрабатываются одинаково. Фреймворт Spring настраивает их, связывает вместе и делает доступными для использования в приложении. Ничего особенного.

Но некоторые компоненты имеют более высокую цель. Реализуя определенные интерфейсы, можно заставить Spring обрабатывать эти компоненты особым образом – как часть самого фреймворка. С помощью *специальных компонентов* можно настроить компоненты, чтобы:

- ❑ принимать участие в процессе создания компонента и жизненном цикле фабрики компонентов, реализуя постобработку конфигурации компонента;
- ❑ загружать информацию о конфигурации из внешних файлов;
- ❑ загружать текстовые сообщения из файлов, включая интернационализированные сообщения;
- ❑ принимать и реагировать на события в приложении, опубликованные другими компонентами и самим контейнером Spring;
- ❑ определять их идентичность в рамках контейнера Spring.

В некоторых случаях эти специализированные компоненты уже имеют реализации, входящие в состав Spring. В других случаях может потребоваться реализовать их интерфейсы самостоятельно.

Начнем изучение специализированных компонентов Spring с постобработки других компонентов после их связывания.

3.5.1. Компоненты постобработки

В главе 1 было продемонстрировано, как определять компоненты в пределах контейнера Spring и как их связывать. Чаще всего нет оснований ожидать, что компоненты будут связаны как-то иначе,

чем определено в XML-файле. XML-файл воспринимается как источник информации о настройках объектов приложения.

Но, как видно на рис. 1.5, фреймворк Spring предлагает две возможности включиться в управление жизненным циклом компонента и просмотреть или изменить его конфигурацию. Это называется *постобработкой*. Из названия можно сделать предположение, что это некая обработка, выполняемая после некоторых событий. Событие постобработки следует за событиями создания и настройки компонента. Интерфейс BeanPostProcessor предоставляет две возможности изменить компонент после его создания и связывания:

```
public interface BeanPostProcessor {  
    Object postProcessBeforeInitialization(  
        Object bean, String name) throws BeansException;  
  
    Object postProcessAfterInitialization(  
        Object bean, String name) throws BeansException;  
}
```

Метод postProcessBeforeInitialization() вызывается непосредственно перед инициализацией компонента (перед вызовом метода afterPropertiesSet() и метода, указанного в атрибуте init-method). Аналогично метод postProcessAfterInitialization() вызывается сразу же после инициализации.

Постпроцессор компонентов

Предположим, для примера, что необходимо изменить все строковые свойства компонентов приложения с целью перевести их на язык Элмера Фудда (Elmer Fudd)¹. Класс Fuddifier, представленный в листинге 3.5, – это реализация интерфейса BeanPostProcessor, которая выполняет именно это действие.

Листинг 3.5. Преобразование строковых свойств с использованием BeanPostProcessor

```
public class Fuddifier implements BeanPostProcessor {  
    public Object postProcessAfterInitialization(  
        Object bean, String name) throws BeansException {
```

¹ Когда все буквы R и L в словах замещаются буквой W. Чтобы получить представление, как это может выглядеть, попробуйте в словах русского языка заменить буквы Р и Л буквой В. Элмер Фудд (яйцеголовый) – мультипликационный герой, заклятый враг кролика Багс Банни (Bugs Bunny). – Прим. перев.

```

Object bean, String name) throws BeansException {
Field[] fields = bean.getClass().getDeclaredFields();

try {
    for(int i=0; i < fields.length; i++) { // Преобразует все
        if(fields[i].getType().equals(          // строковые свойства
            java.lang.String.class)) {         // компонентов
            fields[i].setAccessible(true);
            String original = (String) fields[i].get(bean);
            fields[i].set(bean, fuddify(original));
        }
    }
} catch (IllegalAccessException e) {
    e.printStackTrace();
}
return bean;
}

private String fuddify(String orig) { // Преобразует все строковые
    if(orig == null) return orig;      // свойства компонентов
    return orig.replaceAll("(r|l)", "w")
        .replaceAll("(R|L)", "W");
}

public Object postProcessBeforeInitialization(           // Перед инициализацией
    Object bean, String name) throws BeansException { // ничего
    return bean;                                     // не делает
}
}

```

Метод `postProcessAfterInitialization()` выполняет в цикле итерации по всем свойствам компонента, выбирая свойства, имеющие тип `java.lang.String`. Каждое строковое свойство передается методу `fuddify()`, преобразующему строку на язык Фудда. В результате в свойстве сохраняется «фуддифицированная» строка текста (обратите внимание также на вызов метода `setAccessible()` для каждого свойства, который используется здесь с целью обойти правила видимости частных свойств; это нарушает принцип инкапсуляции, но иначе эти свойства останутся недоступны).

Метод `postProcessBeforeInitialization()` специально оставлен бесполезным – он просто возвращает компонент в неизменном виде. На самом деле «фуддификацию» строковых свойств можно было бы выполнить в этом методе.

Теперь, когда у нас есть постпроцессор BeanPostProcessor, выполняющий преобразование строковых свойств, посмотрим, как сообщить контейнеру, чтобы он применил его ко всем компонентам.

Регистрация постпроцессора компонентов

Если приложение выполняется в рамках фабрики компонентов, каждый постпроцессор BeanPostProcessor придется регистрировать программно, с помощью метода addBeanPostProcessor() фабрики:

```
BeanPostProcessor fuddifier = new Fuddifier();
factory.addBeanPostProcessor(fuddifier);
```

Более вероятно, однако, что вы предпочтете использовать контекст приложения. В этом случае достаточно просто зарегистрировать постпроцессор как компонент внутри контекста.

```
<bean
    class="com.springinaction.chapter03.postprocessor.Fuddifier"/>
```

Контейнер опознает компонент fuddifier как реализацию интерфейса BeanPostProcessor и будет вызывать его методы постобработки до и после инициализации каждого компонента.

В результате регистрации компонента fuddifier все строковые свойства всех компонентов будут «фудифицированы». Например, допустим, что имеется следующее определение компонента:

```
<bean id="bugs" class="com.springinaction.chapter03.postprocessor.Rabbit">
    <property name="description" value="That rascally rabbit!" />
</bean>
```

Когда постпроцессор fuddifier завершит обработку, свойство description будет содержать строку «That wascawwy wabbit!»¹.

Собственные постпроцессоры фреймворка Spring

Фреймворк Spring Framework сам использует несколько реализаций интерфейса BeanPostProcessor. Например, ApplicationContextAwareProcessor, устанавливающий контекст приложения в компонентах, реализующих интерфейс ApplicationContextAware (см. раздел 3.5.6). Его

¹ Или на русском языке: «Этот подлый кролик!» – «Этот подвый квовик!» – Прим. перев.

не нужно регистрировать, он будет зарегистрирован автоматически, контекстом приложения.

В главе 5 будет показана другая реализация интерфейса BeanPostProcessor.

3.5.2. Постобработка контейнера

Реализация интерфейса BeanPostProcessor выполняет постобработку компонента после его загрузки, тогда как реализация интерфейса BeanFactoryPostProcessor выполняет постобработку всего контейнера Spring. Интерфейс BeanFactoryPostProcessor определен следующим образом:

```
public interface BeanFactoryPostProcessor {  
    void postProcessBeanFactory(  
        ConfigurableListableBeanFactory beanFactory)  
        throws BeansException;  
}
```

Метод postProcessBeanFactory() вызывается контейнером Spring после загрузки определений всех компонентов, но перед созданием любых их экземпляров (включая компоненты BeanPostProcessor).

Например, следующая реализация интерфейса BeanFactoryPostProcessor придает новый смысл термину «счетчик компонентов»:

```
public class BeanCounter implements BeanFactoryPostProcessor {  
    private Logger LOGGER = Logger.getLogger(BeanCounter.class);  
    public void postProcessBeanFactory(  
        ConfigurableListableBeanFactory factory)  
        throws BeansException {  
        LOGGER.debug("BEAN COUNT: " +  
            factory.getBeanDefinitionCount());  
    }  
}
```

Класс BeanCounter реализует интерфейс BeanFactoryPostProcessor и просто записывает количество компонентов, загруженных в фабрику компонентов.

При использовании контейнера контекста приложения, чтобы зарегистрировать реализацию интерфейса BeanFactoryPostProcessor, достаточно просто объявить ее как обычный компонент:

```
<bean id="beanCounter"  
      class="com.springinaction.chapter03.postprocessor.BeanCounter"/>
```

Когда контейнер обнаружит, что компонент `beanCounter` реализует интерфейс `BeanFactoryPostProcessor`, он автоматически зарегистрирует его для постобработки фабрики компонентов. Реализации интерфейса `BeanFactoryPostProcessor` нельзя использовать совместно с базовыми контейнерами фабрики компонентов – эта возможность доступна только в контейнерах контекста приложения.

Класс `BeanCounter` в данном примере является достаточно простой реализацией интерфейса `BeanFactoryPostProcessor`. Чтобы найти более сложные примеры, не нужно идти слишком далеко. Вы уже видели `CustomerEditorConfigurer`, реализующий интерфейс `BeanFactoryPostProcessor` и используемый для регистрации пользовательских редакторов свойств в Spring.

Еще одна достаточно полезная реализация `BeanFactoryPostProcessor` – класс `PropertyPlaceholderConfigurer`. Он загружает свойства из одного или более внешних файлов свойств и использует их для подстановки значений переменных в XML-файле конфигурации компонентов. Класс `PropertyPlaceholderConfigurer` будет рассмотрен далее.

3.5.3. Внешние файлы с настройками свойств

Практически всегда все настройки приложения можно сосредоточить в одном XML-файле. Но иногда может оказаться полезным перенести некоторую их часть в отдельный файл. Например, для многих приложений общей частью конфигурации является конфигурация источника данных. В Spring источник данных можно настроить, как показано ниже:

```
<bean id="dataSource" class=
    "org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="url"
        value="jdbc:hsqldb:Training" />
    <property name="driverClassName"
        value="org.hsqldb.jdbcDriver" />
    <property name="username" value="appUser" />
    <property name="password" value="password" />
</bean>
```

Включение настроек источника данных в общий файл с описанием компонентов может оказаться нецелесообразным. Параметры подключения к базе данных определяются на этапе развертывания

приложения. Тогда как файл с описанием компонентов определяет порядок их связывания внутри приложения. Это не означает, что нельзя настроить все компоненты приложения в одном файле. В самом деле, когда конфигурация отражает специфику конкретного приложения (а не специфику его развертывания), настройки компонентов предпочтительнее хранить в ее отдельном файле, чтобы отделить их от настроек, определяемых на этапе развертывания.

К счастью, вынести определение свойств в отдельный файл достаточно легко, если в качестве контейнера Spring использовать ApplicationContext. Чтобы сообщить фреймворку Spring о необходимости загружать конфигурацию из внешнего файла свойств, можно воспользоваться классом PropertyPlaceholderConfigurer. Для этого необходимо настроить следующий компонент в файле связывания компонентов:

```
<bean id="propertyConfigurer"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="jdbc.properties" />
</bean>
```

Свойство location позволяет работать с одним файлом свойств. Если потребуется разбить конфигурацию свойств на несколько файлов, следует использовать свойство locations класса PropertyPlaceholderConfigurer и указать в нем список файлов со свойствами:

```
<bean id="propertyConfigurer"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
      <list>
        <value>jdbc.properties</value>
        <value>security.properties</value>
        <value>application.properties</value>
      </list>
    </property>
</bean>
```

Теперь можно заменить жестко определенные значения в файле с описанием компонентов соответствующими переменными. Синтаксически переменные записываются в форме \${variable}, аналогично тому, как описываются свойства в языке утилиты Ant и в языке выражений JSP. После подстановки переменных новое определение компонента dataSource будет выглядеть так:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="url" value="${database.url}" />
    <property name="driverClassName"
              value="${database.driver}" />
    <property name="username"
              value="${database.user}" />
    <property name="password"
              value="${database.password}" />
</bean>
```

Когда фреймворк Spring создаст компонент источника данных dataSource, в работу включится объект PropertyPlaceholderConfigurer и заменит переменные их значениями из файла со свойствами, как показано на рис. 3.7.

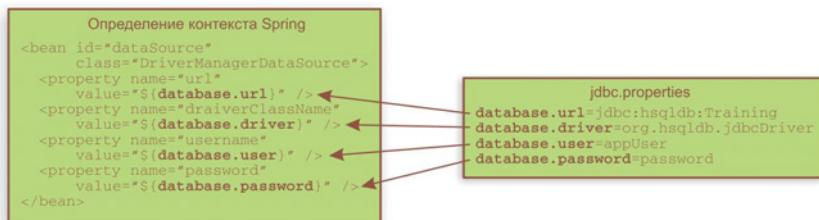


Рис. 3.7. PropertyPlaceholderConfigurer позволяет выносить значения параметров настройки в отдельный файл со свойствами и затем загружать их для подстановки на место переменных в описании контекста Spring

Класс PropertyPlaceholderConfigurer удобно использовать для выделения части настроек в отдельный файл свойств. Но Java использует файлы свойств не только для конфигурации, они также часто используются для хранения текстовых сообщений и интернационализации. Посмотрим, как можно организовать извлечение текстовых сообщений из файлов свойств.

3.5.4. Извлечение текстовых сообщений

Часто бывает нежелательно жестко определять некоторый текст, который будет отображаться перед пользователем приложения. Причиной тому может быть необходимость изменения текста с течением времени, или, возможно, приложение должно поддерживать интернационализацию и отображать текст на родном языке пользователя.

Поддержка параметризации и интернационализации сообщений в Java позволяет определить один или несколько файлов с описаниями свойств, содержащими текст для отображения в приложении. Вместе с файлами свойств, содержащими сообщения на других языках, в приложении всегда должен иметься файл с сообщениями, используемый по умолчанию. Например, если пакет сообщений для приложения называется «*trainingtext*», то в приложении может присутствовать следующий набор файлов с сообщениями:

- ❑ *trainingtext.properties* – сообщения по умолчанию, когда языковые не определены или когда для конкретного языка отсутствует файл со свойствами;
- ❑ *trainingtext_en_US.properties* – сообщения для пользователей в США, говорящих на английском языке;
- ❑ *trainingtext_es_MX.properties* – сообщения для пользователей в Мексике, говорящих на испанском языке;
- ❑ *trainingtext_de_DE.properties* – сообщения для пользователей в Германии, говорящих на немецком языке.

Например, файлы с сообщениями по умолчанию и на английском языке могут содержать следующие записи:

```
course=class
student=student
computer=computer
```

а файл с сообщениями на испанском языке:

```
course=clase
student=estudiante
computer=computadora
```

Контекст приложения Spring поддерживает параметризацию сообщений, обеспечивая доступ к ним через интерфейс *MessageSource*:

```
public interface MessageSource {
    String getMessage(
        MessageSourceResolvable resolvable, Locale locale)
        throws NoSuchMessageException;
    String getMessage(
        String code, Object[] args, Locale locale)
        throws NoSuchMessageException;
    String getMessage(
```

```
        String code, Object[] args, String defaultMessage,  
        Locale locale);  
    }
```

В состав фреймворка Spring входит готовая к использованию реализация интерфейса `MessageSource`. Класс `ResourceBundleMessageSource` просто извлекает сообщения, используя для этого уже имеющийся в Java класс `java.util.ResourceBundle`. Чтобы воспользоваться классом `ResourceBundleMessageSource`, надо добавить следующее определение компонента:

```
<bean id="messageSource"  
      class="org.springframework.context.support.ResourceBundleMessageSource">  
  
    <property name="basename">  
      <value>trainingtext</value>  
    </property>  
</bean>
```

Очень важно, чтобы данный компонент назывался именно `messageSource`, потому что `ApplicationContext` будет искать его по имени, при настройке внутреннего источника сообщений. Зато вам не придется явно внедрять компонент `messageSource` в компоненты приложения. Вместо этого вы получите доступ к сообщениям через метод `getMessage()` контекста приложения `ApplicationContext`. Например, получить сообщение с именем `computer` можно так:

```
Locale locale = ... ; // определить языковые настройки  
String text = context.getMessage("computer", new Object[0], locale);
```

Вам наверняка придется использовать параметризованные сообщения в контексте веб-приложения, чтобы отобразить их на веб-странице. В этом случае для извлечения сообщений можно использовать специальный JSP-тег `<spring:message>`, и тогда не потребуется непосредственно обращаться к `ApplicationContext`:

```
<spring:message code="computer" />
```

Но если извлекать сообщения потребуется в компонентах, а не в JSP, то как получить доступ к `ApplicationContext`? Для этого надо немного подождать или сразу перейти к разделу 3.5.6, где я расскажу, как обеспечить доступность контейнера в компонентах.

А сейчас перейдем к анализу событий, происходящих в течение жизненного цикла контекста приложения, и поговорим о том, как использовать эти события для выполнения специальных операций. Вы также увидите, как можно публиковать собственные события, чтобы обеспечить взаимодействие между разными, никак не связанными между собой компонентами.

3.5.5. Уменьшение связности с использованием событий

Внедрение зависимостей является в Spring основным способом ослабления связей между прикладными объектами, но это не единственный путь. Еще одним способом взаимодействия объектов являются публикация и обработка событий. Генерируя события, объекты могут взаимодействовать с другими объектами, даже не зная, какие объекты принимают их. Более того, объект, принимающий события, может реагировать на них, не имея представления о том, кто генерирует эти события.

Такие взаимодействия, основанные на событиях, напоминают радиостанцию и ее аудиторию слушателей, как показано на рис. 3.8. Радиоприемники не подключены непосредственно к радиостанции, и радиостанция не знает о настройках радиоприемников. Тем не

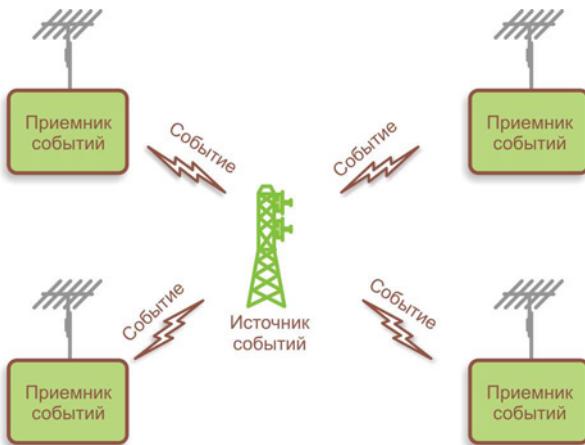


Рис. 3.8. Источник событий напоминает радиостанцию, вещающую о событиях для своих слушателей. Источник событий и приемники полностью отделены друг от друга

менее радиостанция оказывается в состоянии вещать передачи для своих слушателей.

В Spring любой компонент в контейнере может быть приемником или источником событий, или и тем, и другим. Посмотрим, как создавать компоненты, участвующие в событиях, и начнем с компонентов, генерирующих (или публикующих) события.

Публикация событий

Представьте, что в системе регистрации колледжа необходимо предусмотреть возможность извещения одного или более объектов приложения, что некоторый студент подписался на курс и, как следствие, курс оказался набран полностью. При этом, возможно, вы захотите автоматически объявить набор на другой курс.

Сначала определим собственное событие, такое как `CourseFullEvent`:

```
public class CourseFullEvent extends ApplicationEvent {  
    private Course course;  
  
    public CourseFullEvent(Object source, Course course) {  
        super(source);  
        this.course = course;  
    }  
  
    public Course getCourse() {  
        return course;  
    }  
}
```

Далее необходимо опубликовать событие. Интерфейс `ApplicationContext` имеет метод `publishEvent()`, позволяющий публиковать события `ApplicationEvents`. Любой приемник событий `ApplicationListener`, зарегистрировавшийся в контексте приложения, будет получать событие в виде вызова метода `onApplicationEvent()`:

```
ApplicationContext context = ...;  
Course course = ...;  
context.publishEvent(new CourseFullEvent(this, course));
```

К сожалению, чтобы иметь возможность публиковать (посыпать) события, компоненты должны иметь доступ к контексту приложения `ApplicationContext`. Это означает, что компоненты должны знать, в каком контейнере они выполняются. Как сделать так, чтобы ком-

поненты знали, в каком контейнере они действуют, будет показано в разделе 3.5.6.

Скажите, что произойдет с событием, сгенерированным источником, которое никто не примет? Я оставляю этот философский вопрос вам на осмысление. А сейчас обеспечим наличие компонентов, принимающих события, чтобы они не остались без внимания.

Прием событий

В дополнение к событиям, которые генерируются компонентами, фреймворк Spring сам генерирует несколько событий в течение времени жизни приложения. Все эти события являются подклассами абстрактного класса `org.springframework.context.ApplicationEvent`. Ниже представлены три таких события:

- ❑ `ContextClosedEvent` – генерируется при закрытии контекста приложения;
- ❑ `ContextRefreshedEvent` – генерируется после инициализации или обновления контекста приложения;
- ❑ `RequestHandledEvent` – генерируется в контексте веб-приложения после обработки запроса.

По большей части эти события генерируются, чтобы... м-м-м... э-э-э.., в общем генерируются. Большинство компонентов понятия не имеют об этих событиях и зачем они были сгенерированы. Но как быть, если компонент должен получать уведомления о событиях в приложении?

Если требуется, чтобы компонент реагировал на события, сгенерированные другим компонентом или контейнером, достаточно просто реализовать интерфейс `org.springframework.context.ApplicationListener`. Для этого следует реализовать метод `onApplicationEvent()`, отвечающий за реакцию на события:

```
public class RefreshListener implements ApplicationListener {  
    public void onApplicationEvent(ApplicationEvent event) {  
        ...  
    }  
}
```

После этого единственное, что осталось сделать, – сообщить фреймворку Spring о готовности компонента принимать события, зарегистрировав его в контексте компонента:

```
<bean id="refreshListener"  
      class="com.springinaction.foo.RefreshListener"/>
```

Когда контейнер загрузит компоненты в контекст приложения, он обнаружит реализацию интерфейса `ApplicationListener` и будет помнить, что надо вызывать его метод `onApplicationEvent()` при появлении события.

Единственное, что следует помнить всегда, – события обрабатываются синхронно. Поэтому следует позаботиться, чтобы любые события обрабатывались быстро. В противном случае это отрицательно скажется на производительности приложения.

Как упоминалось выше, чтобы генерировать события, компонент должен знать о контейнере приложения. Даже если объект не генерирует никаких событий, все равно может потребоваться разработать компонент, обладающий доступом к контексту приложения. Посмотрим далее, как создавать компоненты, со ссылками на их контейнер.

3.5.6. Создание «осведомленных» компонентов

Вы видели фильм «Матрица»? В этом фильме люди были невольно порабощены машинами, жили своими каждодневными жизнями в виртуальном мире, в то время как суть их жизни состояла в том, чтобы обеспечить машины энергией. Главному персонажу Томасу Андерсону (Thomas Anderson) был дан выбор – принять красную пиллюю и узнать всю правду о своем существовании или принять синюю пиллюю и продолжить прежнюю жизнь. Он выбрал красную и сделался осведомленным о своей реальной личности и о виртуальном мире.

Большая часть компонентов, выполняющихся в контейнере Spring, подобны людям в фильме «Матрица». Для компонентов не знание есть счастье. Они (компоненты) не знают (им даже не положено знать) свои имена или что они действуют внутри контейнера Spring. Как правило, это хорошо, потому что когда компонент знает о контейнере, его связь со Spring усиливается, и он уже не сможет существовать вне контейнера.

Но иногда компонентам надо знать больше. Иногда им надо знать, кто они есть и где они действуют. Иногда им необходимо взять красную пиллюю.

Красная пиллюя в случае с компонентами – это интерфейсы `NameAware`, `BeanFactoryAware` и `ApplicationContextAware`. Реализуя эти три интерфейса, компоненты будут знать свое имя и свой контейнер – `BeanFactory` или `ApplicationContext` соответственно.

Но имейте в виду, что, реализуя эти интерфейсы, компоненты становятся связанными со Spring. И в зависимости от того, как компонент использует это знание, у вас может не быть возможности использовать его вне Spring.

Кто ты есть

Контейнер Spring сообщает компоненту о его имени через интерфейс BeanNameAware. Этот интерфейс определяет единственный метод setBeanName(), который получает строку с именем компонента, указанную в атрибуте id элемента <bean> в файле с настройками компонентов:

```
public interface BeanNameAware {  
    void setBeanName(String name);  
}
```

Знание своего имени может пригодиться компоненту для ведения учета. Например, компонент может существовать в контексте приложения в нескольких экземплярах, и возможность самоидентификации по имени и типу для регистрации в журнале собственных действий может оказаться весьма кстати.

Внутри самого фреймворка Spring Framework интерфейс BeanNameAware используется в разных целях. Наиболее известным примером могут служить компоненты, выполняющие функции планирования. Например, CronTriggerBean, реализующий интерфейс BeanNameAware для установки имени своего задания CronTrigger. Это иллюстрирует следующий отрывок из CronTriggerBean:

```
package org.springframework.scheduling.quartz;  
public class CronTriggerBean extends CronTrigger  
    implements ..., BeanNameAware, ... {  
    ...  
    private String beanName;  
    ...  
    public void setBeanName(String beanName) {  
        this.beanName = beanName;  
    }  
    ...  
    public void afterPropertiesSet() ... {  
        if (getName() == null){  
            setBeanName(this.beanName);  
        }  
        ...  
    }  
    ...  
}
```

Не требуется ничего особенного, чтобы контейнер Spring вызвал метод `setBeanName()` в классе, реализующем интерфейс `BeanNameAware`. Во время загрузки компонента контейнер обнаружит, что компонент реализует интерфейс `BeanNameAware`, и автоматически вызовет метод `setBeanName()`, передав ему имя компонента, объявленное в атрибуте `id` (либо `name`) элемента `<bean>` в XML-файле конфигурации компонентов.

Здесь класс `CronTriggerBean` расширяет класс `CronTrigger`. После того как контекст Spring установит значения всех свойств компонента, он вызовет метод `setBeanName()` (объявленный в `CronTriggerBean`) и передаст ему имя компонента, которое будет использоваться для установки имени планируемого задания.

Этот пример иллюстрирует использование интерфейса `BeanNameAware` на примере встроенного в Spring планировщика. Подробнее о планировании будет рассказано в главе 17. А пока посмотрим, как наделить компонент знанием о контейнере Spring, внутри которого он действует.

Где ты живешь

Как было показано в этом разделе выше, иногда бывает полезно иметь в компоненте доступ к контексту приложения. Компоненту может потребоваться иметь доступ к параметризованному тексту сообщения. Или, возможно, ему необходимо иметь возможность генерировать события для приемников, отвечающих на них. Независимо от причин компоненту может потребоваться знать о контейнере, в котором он выполняется.

Получить информацию о контейнере компоненту помогут интерфейсы `ApplicationContextAware` и `BeanFactoryAware`. Эти интерфейсы объявляют методы `setApplicationContext()` и `setBeanFactory()` соответственно. Контейнер Spring автоматически обнаруживает компоненты, реализующие любой из этих интерфейсов, и обеспечит передачу компоненту ссылки на соответствующий контейнер.

Имея доступ к контексту приложения `ApplicationContext`, компонент может активно взаимодействовать с контейнером. Это может пригодиться для программного поиска зависимостей в контейнере (если они не были внедрены самим контейнером) или для публикации событий в приложении. Возвращаясь к примеру публикации событий, приведенному выше, можно завершить его, как показано ниже:

```
public class StudentServiceImpl  
    implements StudentService, ApplicationContextAware {
```

```
private ApplicationContext context;

public void setApplicationContext(ApplicationContext context) {
    this.context = context;
}

public void enrollStudentInCourse(Course course, Student student)
    throws CourseException;
...
context.publishEvent(new CourseFullEvent(this, course));
...
}
```

Осведомленность о контейнере приложения – одновременно и благо, и проклятие для компонента. С одной стороны, доступ к контексту приложения расширяет возможности компонента, с другой стороны – знание о контейнере связывает компонент с фреймворком Spring, а это то, чего, в принципе, желательно избегать, по возможности.

До сих пор предполагалось, что все компоненты в контейнере Spring реализованы как Java-классы. Это вполне обоснованное предположение, но совершенно необязательное. Посмотрим далее, как добавить в приложение динамическое поведение, включая в него компоненты, реализованные с использованием языков сценариев.

3.6. Компоненты, управляемые сценариями

Программный код на языке Java, который превращается в компоненты приложения на основе Spring, в конечном итоге компилируется в байт-код и выполняется виртуальной машиной JVM. Более того, вы наверняка упакуете скомпилированный код в JAR-, WAR- или EAR-файл для последующего развертывания. Но что, если после развертывания приложения потребуется изменить поведение кода?

Проблема со статически скомпилированным кодом состоит в том, что он... таки... статичен. Скомпилированный в файл класса и упакованный в дистрибутив, его сложно будет изменить без перекомпиляции, повторной упаковки и развертывания всего приложения. Во многих обстоятельствах это приемлемо (и даже желательно). Но

и в этом случае статически скомпилированный код не позволяет быстро реагировать на изменяющиеся потребности бизнеса.

Предположим, например, что вы разрабатываете приложение для электронной коммерции. В приложении имеется программный код, вычисляющий промежуточные и общую сумму покупки на основе выбранных позиций в заказе, тарифов доставки и налогов с продажи. Но что, если потребуется не учитывать налоги в течение одного дня?

Если часть приложения, вычисляющая налоговую составляющую, скомпилирована статически вместе с остальным кодом, единственным доступным вариантом будет развертывание «безналоговой» версии приложения в полночь, с последующим развертыванием базовой версии в следующую полночь. Вам придется заготовить кофе на две ночи.

В приложениях на основе фреймворка Spring имеется возможность писать программный код на языке Ruby, Groovy или BeanShell и внедрять его в контекст приложения, как если бы это был обычный компонент, написанный на языке Java, как показано на рис. 3.9. В следующих нескольких подразделах я продемонстрирую, как писать сценарии на языках Ruby, Groovy и BeanShell для компонентов, выполняющихся под управлением Spring.

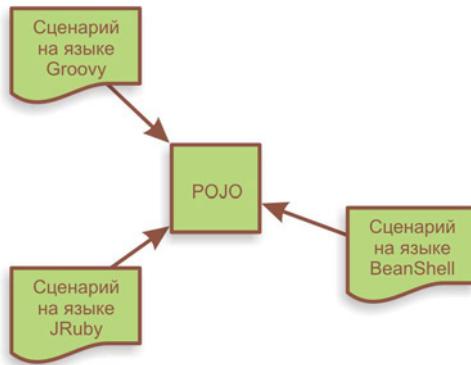


Рис. 3.9. Фреймворк позволяет внедрять не только POJO в POJO. Он предоставляет возможность динамически модифицировать поведение приложения, внедряя в POJO компоненты, управляемые сценариями

Тех, кому нравится музыка Calypso, ожидает сюрприз – я собираюсь продемонстрировать особенности компонентов, управляемых сценариями, проведя аналогии с одной из самых известных песен

этого жанра. Следите за тем, как я добавлю динамически управляемый лайм в кокос Java.

3.6.1. Добавляем лайм в кокос

Для иллюстрации возможности динамического управления компонентов в Spring внедрим реализацию интерфейса `Lime`, управляемую сценарием, в Java-объект `Coconut`. Для начала познакомимся с классом `Coconut`, представленным в листинге 3.6.

Листинг 3.6. Java-класс в кокосовой скорупе

```
package com.springinaction.scripting;

public class Coconut {
    public Coconut() {}

    public void drinkThemBothUp() {
        System.out.println("You put the lime in the coconut...");
        System.out.println("and drink 'em both up...");
        System.out.println("You put the lime in the coconut...");
        lime.drink(); // Вызов метода drink() интерфейса Lime
    }

    // Внедряемый компонент
    private Lime lime; // Для внедрения
    public void setLime(Lime lime) { // реализации
        this.lime = lime; // интерфейса Lime
    }
}
```

Класс `Coconut` имеет один простой метод с именем `drinkThemBothUp()`. При вызове этого метода в поток `System.out` выводится начало четверостишья из песни мистера Нильссона (`Mr. Nilsson`). В последней строке метода вызывается метод `drink()` внедренного объекта `Lime`, который выводит последнюю строку четверостишья. Внедренный компонент – это произвольный объект, реализующий следующий интерфейс:

```
package com.springinaction.scripting;

public interface Lime {
    void drink();
}
```

В процессе конфигурирования приложения в класс `Coconut` внедряется ссылка на объект `Lime`, как показано в следующем фрагменте XML-кода:

```
<bean id="coconut" class="com.springinaction.scripting.Coconut">
    <property name="lime" ref="lime" />
</bean>
```

Пока ни в классе `Coconut`, ни в интерфейсе `Lime` не видно ничего особенного, что указывало бы на компоненты, управляемые сценариями. Этот программный код напоминает примеры простых компонентов JavaBean и внедрения зависимостей (DI) из главы 2.

Единственное, чего здесь не хватает, – реализации интерфейса `Lime` и ее объявления в контексте Spring. Здесь подойдет любая реализация интерфейса `Lime` на языке Java. Но я обещал лайм, управляемый сценарием, поэтому далее я представлю вам такой лайм.

3.6.2. Компонент, управляемый сценарием

Реализацию интерфейса `Lime`, управляемую сценарием, можно выполнить на языке Ruby, Groovy или BeanShell. Но независимо от выбора языка сначала необходимо добавить некоторые настройки в файл определения контекста Spring. Следующее объявление `<beans>` демонстрирует, как это делается:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:lang="http://www.springframework.org/schema/lang"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://www.springframework.org/schema/lang
        http://www.springframework.org/schema/lang/
        ↳spring-lang-2.0.xsd">
    ...
</beans>
```

В состав Spring 2.0 входят несколько новых конфигурационных элементов, каждый из которых определен в собственном пространстве имен и в схеме XML. В течение книги нам не раз еще будут встречаться дополнительные пространства имен, имеющиеся в Spring. А пока достаточно будет сказать, что выделенный фрагмент в этом объявлении `<beans>` сообщает фреймворку Spring, что

далее будут использоваться некоторые элементы из пространства имен lang.

Теперь, когда пространство имен объявлено в конфигурационном файле контекста Spring, можно приступать к созданию реализации интерфейса Lime, управляемой сценарием. Начнем с рубинового лайма.

Рубиновый лайм

В последние годы язык Ruby привлек внимание многих разработчиков приложений на языке Java, поэтому никого не удивит, если вы пожелаете реализовать компонент, управляемый сценарием на этом весьма популярном языке сценариев. Ниже представлен сценарий на языке Ruby, реализующий интерфейс Lime и его метод drink():

```
class Lime
  def drink
    puts "Called the doctor woke him up!"
  end
end
Lime.new
```

Обратите внимание, что последняя строка сценария создает новый объект Lime. Эта строка играет важную роль – она создает объект Lime, который затем можно будет внедрить в другие объекты Spring.

Внедрение объекта Lime на языке Ruby выполняется с помощью элемента <lang:jruby>, как показано ниже:

```
<lang:jruby id="lime">
  script-source="classpath:com/springinaction/scripting/Lime.rb"
  script-interfaces="com.springinaction.scripting.Lime" />
```

В элементе <lang:jruby> имеются два обязательных атрибута. Первый, script-source, сообщает фреймворку Spring, где находится файл сценария. В данном случае файл Lime.rb находится в библиотеке классов (classpath), в том же пакете, что и остальной программный код примера. Второй атрибут, script-interfaces, сообщает, какой Java-интерфейс реализует этот сценарий. В данном случае он реализует наш интерфейс Lime.

Компонент, управляемый сценарием на языке Groovy

Несмотря на огромную популярность языка Ruby, многие разработчики предпочитают оставаться на платформе Java. Groovy – это

язык программирования, соединяющий в себе лучшие черты Ruby и других языков сценариев и знакомый синтаксис языка Java. Фактически для Java-разработчиков он является примером сочетания всего самого лучшего из двух миров.

Для тех, кто отдаст предпочтение языку Groovy, ниже приводится реализация интерфейса `Lime` на языке Groovy:

```
class Lime implements com.springinaction.scripting.Lime {  
    void drink() {  
        print "Called the doctor woke him up!"  
    }  
}
```

Внедрение сценария на языке Groovy в компонент Spring выполняется с помощью элемента `<lang:groovy>`. Следующий элемент `<lang:groovy>` загрузит реализацию интерфейса `Lime` на языке Groovy:

```
<lang:groovy id="lime"  
    script-source="classpath:com/springinaction/scripting/Lime.groovy" />
```

Как и в элементе `<lang:jruby>`, атрибут `script-source` определяет местоположение файла со сценарием на языке Groovy. В данном случае файл сценария также располагается в библиотеке классов (`classpath`), в том же пакете, что и остальной код примера.

Однако, в отличие от элемента `<lang:jruby>`, элемент `<lang:groovy>` не требует (и даже не поддерживает) указывать атрибут `script-interfaces`. Это объясняется тем, что сценарий на языке Groovy содержит достаточно информации, чтобы определить, какие интерфейсы он реализует. Обратите внимание, что Groovy-класс `Lime` явно реализует интерфейс `com.springinaction.scripting.Lime`.

Компонент, управляемый сценарием на языке BeanShell

Фреймворк поддерживает еще один язык сценариев – BeanShell. В отличие от Ruby и Groovy, имеющих собственный синтаксис, язык BeanShell имитирует синтаксис языка Java, что делает его привлекательным для тех, кто собирается встраивать сценарии в приложение, но не желает тратить время на изучение других языков.

В заключение обзора языков сценариев, поддерживаемых фреймворком Spring, ниже приводится реализация интерфейса `Lime` на языке BeanShell:

```
void drink() {
    System.out.println("Called the doctor woke him up!");
}
```

Первое, что бросается в глаза в этом сценарии, – в нем отсутствует определение класса, присутствует только метод `drink()`. В сценариях на языке BeanShell определяются только методы, требуемые интерфейсом, но не класс.

Внедрение сценария на языке BeanShell выполняется точно так же, как и сценария на языке Ruby, за исключением того, что в данном случае используется элемент `<lang:bsh>`, как показано ниже:

```
<lang:bsh id="lime"
  script-source="classpath:com/springinaction/scripting/Lime.bsh"
  script-interfaces="com.springinaction.scripting.Lime" />
```

Как и в предыдущих элементах объявления сценария, атрибут `script-source` определяет местоположение файла сценария. И, подобно элементу `<lang:jruby>`, атрибут `script-interfaces` определяет интерфейсы, реализованные в сценарии.

Теперь вы знаете, как настраивать компоненты, управляемые сценариями, и как внедрять их в свойства POJO. Но что, если потребуется выполнить внедрение в обратном направлении? Посмотрим, как внедрить POJO в компонент, управляемый сценарием.

3.6.3. Внедрение в свойства компонентов, управляемых сценариями

Чтобы проиллюстрировать, как реализовать внедрение значений в свойства компонента, управляемого сценарием, перевернем наш пример с лаймом и кокосом с ног на голову. На этот раз кокосом будет компонент, управляемый сценарием, а лаймом – POJO. Сначала рассмотрим Java-класс, реализующий интерфейс `Lime`:

```
package com.springinaction.scripting;

public class LimeImpl implements Lime {
    public LimeImpl() {}

    public void drink() {
        System.out.println("Called the doctor woke him up!");
    }
}
```

LimeImpl – это обычный Java-класс, реализующий интерфейс Lime. А ниже приводится его конфигурация в Spring:

```
<bean id="lime" class="com.springinaction.scripting.LimeImpl" />
```

Пока ничего особенного. Теперь реализуем класс Coconut как сценарий на языке Groovy:

```
class Coconut implements com.springinaction.scripting.ICoconut {  
    public void drinkThemBothUp() {  
        println "You put the lime in the coconut..."  
        println "and drink 'em both up..."  
        println "You put the lime in the coconut..."  
        lime.drink()  
    }  
    com.springinaction.scripting.Lime lime;  
}
```

Подобно Java-версии, этот класс Coconut выводит первые строки четверостишия и затем вызывает метод drink() свойства lime, завершающий это четверостишие. Здесь свойство lime объявлено как некоторая реализация интерфейса Lime.

Теперь осталось лишь сконфигурировать компонент Coconut и внедрить в него компонент lime:

```
<lang:groovy id="coconut"  
    script-source="classpath:com/springinaction/scripting/Coconut.groovy">  
    <lang:property name="lime" ref="lime" />  
</lang:groovy>
```

Здесь компонент Coconut, управляемый сценарием, объявлен точно так же, как в предыдущем разделе был объявлен компонент Lime. Но, помимо элемента `<lang:groovy>`, здесь также присутствует элемент `<lang:property>`, помогающий выполнить внедрение зависимости.

Элемент `<lang:property>` можно использовать в описаниях любых компонентов, управляемых сценариями. Он практически идентичен элементу `<property>`, с которым вы познакомились в главе 2, за исключением того, что он предназначен для внедрения значений в свойства компонентов, управляемых сценариями, а не в свойства POJO.

В данном случае элемент `<lang:property>` связывает свойство lime компонента coconut со ссылкой на компонент lime, который в этом

примере является компонентом JavaBean. Кому-то может показаться интересной возможность внедрения компонентов, управляемых сценариями, в свойства других компонентов, также управляемых сценариями. В действительности вполне возможно внедрить компонент со сценарием на языке BeanShell в компонент со сценарием на языке Groovy, который, в свою очередь, внедрен в компонент со сценарием на языке Ruby. Более того, хоть это и выглядит слишком экстремально, на языках сценариев теоретически возможно написать все приложение!

3.6.4. Обновление компонентов, управляемых сценариями

Одним из основных преимуществ использования сценариев, вместо статически компилируемого программного кода на языке Java, является возможность их изменения без необходимости повторно разворачивать приложение. Если бы реализация интерфейса Lime была написана на языке Java и вы решили бы изменить четверостишье, которое она выводит, вам пришлось бы заново скомпилировать реализацию класса и повторно развернуть приложение. Но при использовании сценариев реализацию можно изменить в любой момент и почти немедленно получить результат.

Это самое «почти немедленно» зависит от того, как часто фреймворк Spring проверяет наличие изменений в сценариях. Все элементы настройки компонентов, управляемых сценариями, имеют атрибут refresh-check-delay, позволяющий определить, как часто (в миллисекундах) сценарий должен обновляться фреймворком.

По умолчанию атрибут refresh-check-delay имеет значение –1, что означает запрет на обновление. Но если представить, что вам требуется обновлять сценарий каждые 5 секунд, тогда можно было бы использовать следующее определение элемента <lang:jruby>:

```
<lang:jruby id="lime">
    script-source="classpath:com/springinaction/scripting/Lime.rb"
    script-interfaces="com.springinaction.scripting.Lime"
    refresh-check-delay="5000"/>
```

Обратите внимание: несмотря на то что в этом примере использован элемент <lang:jruby>, атрибут refresh-check-delay действует одинаково и в элементах <lang:groovy> и <lang:bsh>.

3.6.5. Создание компонентов, управляемых сценариями, непосредственно в конфигурации

Обычно сценарии для компонентов принято сохранять во внешних файлах и ссылаться на них с использованием атрибута `script-source`. Однако в некоторых случаях бывает удобно сохранять программный код сценария непосредственно в конфигурационном файле Spring.

Для этого все элементы объявления сценариев поддерживают включение в них вложенного элемента `<lang:inline-script>`. Например, следующий фрагмент XML-кода объявляет компонент `lime`, управляемый сценарием на языке BeanShell, непосредственно в конфигурационном файле Spring:

```
<lang:bsh id="lime">
    script-interfaces="com.springinaction.scripting.Lime">
        <lang:inline-script><![CDATA[
            void drink() {
                System.out.println("Called the doctor woke him up!");
            }
        ]]>
    </lang:inline-script>
</lang:bsh>
```

Вместо атрибута `script-source` в этом определении компонента `lime` используется элемент `<lang:inline-script>`, содержащий сценарий на языке BeanShell.

Отметьте, что здесь программный код сценария заключен в элемент `<![CDATA[...]]>`. Сценарий может содержать символ или текст, который по ошибке может быть воспринят как код разметки XML. Конструкция `<![CDATA[...]]>` предотвращает это. В данном примере сценарий не содержит ничего, что могло бы ввести в заблуждение парсер XML. Тем не менее лучше всегда использовать элемент `<![CDATA[...]]>`, чтобы не попасть впросак в будущем, когда придется изменить сценарий.

В этом разделе вы познакомились с несколькими элементами конфигурации фреймворка Spring, выходящими за пределы элементов `<bean>` и `<property>`, представленные в главе 2. Эти элементы являются лишь некоторыми из новых конфигурационных элементов, появившихся в версии Spring 2.0. По мере чтения книги вы познакомитесь с еще большим количеством конфигурационных элементов.



3.7. В заключение

Основной функцией Spring является связывание объектов приложения между собой. В главе 2 были показаны основные приемы связывания компонентов, используемые в повседневной разработке, а в этой главе мы познакомились с дополнительными, редко используемыми приемами.

Для уменьшения объема повторяющегося XML-кода при определении похожих компонентов фреймворк Spring предоставляет возможность объявления абстрактных компонентов, описывающих общие свойства, и затем определения «дочерних» компонентов от абстрактного компонента.

Пожалуй, одной из самых необычных является возможность фреймворка Spring изменять функциональность компонента посредством приема внедрения методов. Используя этот прием, можно замещать имеющиеся методы новыми их реализациями. В противовес приему внедрения через методы записи существует возможность внедрения методов чтения, когда имеющийся метод чтения можно заменить другим методом чтения, возвращающим определенный компонент.

Не все объекты приложения создаются или управляются фреймворком Spring. Чтобы обеспечить возможность внедрения зависимостей для таких объектов, предоставляется возможность объявлять объекты «настраиваемыми фреймворком Spring». Такие компоненты подхватываются фреймворком после их создания и настраиваются на основе шаблонов компонентов Spring.

Фреймворк Spring использует редакторы свойств для присваивания значений, указанных в конфигурации, благодаря чему даже такие сложные объекты, как URL-адреса или массивы, могут быть настроены с использованием строковых значений. В этой главе было показано, как создавать собственные редакторы, чтобы упростить настройку сложных свойств.

Иногда компонентам требуется взаимодействовать с контейнером Spring. Для этого фреймворком предоставляется несколько интерфейсов, позволяющих компонентам выполнять постобработку, получать свойства из внешних файлов конфигурации, обрабатывать события и даже узнавать собственные имена.

Наконец, для любителей динамических языков сценариев, таких как Ruby, Groovy и BeanShell, фреймворк Spring позволяет создавать компоненты, управляемые сценариями на этих языках. Эта

возможность обеспечивает динамическое поведение приложения, делая возможным «горячую» замену определения компонента, реализованного на одном из трех языков сценариев.

Теперь вы знаете, как связывать объекты в контейнере Spring и как DI помогает ослаблять связи между ними. Но DI – это лишь один из способов обеспечения слабой связанности объектов в Spring. В следующей главе будет показано, как поддержка аспектно-ориентированного программирования в Spring помогает выносить общую функциональность из прикладных объектов.



Глава 4. Сокращение размера XML-конфигурации Spring

В этой главе рассматриваются следующие темы:

- ❑ автоматическое связывание компонентов;
- ❑ автоматическое определение компонентов;
- ❑ связывание компонентов посредством аннотаций;
- ❑ конфигурирование Spring в программном коде на Java.

До сих пор рассматривались приемы объявления компонентов с помощью элементов `<bean>` и внедрения значений в свойства компонентов с помощью элементов `<constructor-arg>` и `<property>`. Эти приемы с успехом могут применяться в небольших приложениях, с малым количеством компонентов. Но в больших приложениях эти приемы способствуют разбуханию XML-файлов, описывающих конфигурацию.

К счастью, фреймворк Spring обладает рядом особенностей, позволяющих уменьшить размер конфигурационных XML-файлов:

- ❑ возможность автоматического связывания зависимостей компонентов помогает уменьшить количество элементов `<property>` и `<constructor-arg>` или даже совсем избавиться от них;
- ❑ возможность автоматического определения компонентов расширяет функцию автоматического связывания, позволяя фреймворку Spring автоматически обнаруживать, какие классы должны настраиваться как компоненты Spring, и устранивая необходимость использовать элементы `<bean>`.

При совместном использовании механизмы автоматического связывания и автоматического определения могут способствовать существенному снижению размеров XML-файлов с конфигурацией Spring. Часто бывает достаточно XML-файла лишь с несколькими строками, независимо от количества компонентов в контексте приложения Spring.

Эта глава начинается с обзора преимуществ механизмов автоматического связывания и автоматического определения в Spring,

позволяющих уменьшить размеры XML-файлов, необходимых для конфигурирования приложений. А заканчивается знакомством с приемами конфигурирования Spring в программном коде на языке Java, опирающимися на старый, добрый Java-код вместо XML.

4.1. Автоматическое связывание свойств компонентов

Если я скажу: «Сегодня луна особенно яркая», – едва ли у кого-то возникнет вопрос: «Какая луна?», – потому что мы с вами жители Земли, и в данном контексте всем очевидно, что я говорю о Луне – единственном спутнике Земли. Если бы мы с вами были жителями Юпитера, вы наверняка захотели бы уточнить, о каком из 63 естественных спутников идет речь. Но на Земле моя фраза звучит более чем однозначно¹. Аналогично, когда дело доходит до автоматического связывания свойств компонентов, очевидно, ссылка на какой компонент должна быть внедрена в данное свойство. Если в контексте приложения имеется только один компонент типа `javax.sql.DataSource`, тогда любой компонент, имеющий свойство типа `DataSource`, будет зависеть *именно от этого* компонента `DataSource`. В конце концов, это единственный компонент такого типа.

Для оформления таких очевидных зависимостей фреймворк Spring предлагает механизм автоматического связывания. Почему бы не позволить фреймворку самому разобраться в подобных ситуациях, когда однозначно можно определить, ссылка на какой компонент должна быть внедрена в свойство, вместо явного связывания свойств?

4.1.1. Четыре типа автоматического связывания

При использовании механизма автоматического связывания фреймворк Spring получает массу подсказок, позволяющих автоматически связать компоненты с их зависимостями. Всего фреймворком поддерживаются четыре разновидности автоматического связывания.

¹ Конечно, если бы мы оказались на Юпитере, яркость его лун едва ли могла заботить нас из-за высокой плотности атмосферы и ее непригодности для дыхания.

- ❑ byName – пытается отыскать компонент в контейнере, чье имя (или идентификатор) совпадает с именем связываемого свойства. Если соответствующий компонент не найден, свойство останется несвязанным.
- ❑ byType – пытается отыскать единственный компонент в контейнере, чей тип соответствует типу связываемого свойства. Если соответствующий компонент не найден, свойство не будет связано.
- ❑ constructor – пытается сопоставить конструктор компонента, куда выполняется внедрение, с компонентами, чьи типы совпадают с аргументами конструктора.
- ❑ autodetect – сначала пытается выполнить автоматическое связывание через конструктор, а затем по типу.

У каждой из этих разновидностей есть свои достоинства и недостатки. Посмотрим сначала, как организовать автоматическое связывание, руководствуясь именами свойств компонентов.

Автоматическое связывание по имени

В Spring все сущее имеет свои имена. Свойствам компонентов, как и самим компонентам, даются определенные имена. Допустим, что имя свойства совпадает с именем компонента, внедряемого в это свойство. Это может служить подсказкой для фреймворка, что данный компонент следует автоматически внедрить в это свойство.

Для примера вернемся к компоненту `kenny`, представленному в главе 2:

```
<bean id="kenny2"
      class="com.springinaction.springidol.Instrumentalist">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

Здесь с помощью элемента `<property>` явно настраивается свойство `instrument`. Представим на мгновение, что `Saxophone` объявлен как элемент `<bean>` с атрибутом `id="instrument"`:

```
<bean id="instrument"
      class="com.springinaction.springidol.Saxophone" />
```

При наличии такого определения идентификатор компонента `Saxophone` совпал бы с именем свойства `instrument` и фреймворк

Spring смог бы использовать это обстоятельство для автоматического связывания инструмента, если бы компонент `kenny` имел атрибут `autowire`, как показано ниже:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist"
      autowire="byName">
    <property name="song" value="Jingle Bells" />
</bean>
```

При автоматическом связывании по имени (`byName`) устанавливается соглашение, в соответствии с которым свойство будет автоматически связано с компонентом, имеющим такое же имя. Атрибут `autowire` со значением `byName` сообщает фреймворку, что необходимо просмотреть все свойства компонента `kenny` и отыскать компоненты с соответствующими им именами. В данном случае свойство `instrument` подходит для автоматического связывания через метод записи. Как изображено на рис. 4.1, если в контексте имеется компонент с идентификатором `instrument`, он будет автоматически внедрен в свойство `instrument`.



Рис. 4.1. При автоматическом связывании по имени имя компонента соответствует свойствам с теми же именами

Недостаток автоматического связывания по имени – в том, что предполагается наличие компонента, чей идентификатор совпадает с именем свойства другого компонента. В нашем примере это потребовало бы создать компонент с именем `instrument`. Если в конфигурации будет определено несколько компонентов типа `Instrumentalist` с атрибутом `autowire`, тогда все они будут играть на одном и том же инструменте. В некоторых ситуациях, возможно, это не проблема, но об этом ограничении следует помнить.

Автоматическое связывание по типу

Автоматическое связывание по типу (`byType`) действует подобно автоматическому связыванию по имени (`byName`), только вместо имени свойства учитывается его тип. При автоматическом связывании

по типу фреймворк Spring будет искать компоненты, чей тип совпадает с типом свойства.

Например, предположим, что атрибут `autowire` компонента `kenny` имеет значение `byType`. Контейнер попытается отыскать внутри себя компонент, имеющий тип `Instrument`, и внедрит его в свойство `instrument`. Как показано на рис. 4.2, компонент `saxophone` будет автоматически внедрен в свойство `instrument` компонента `kenny`, потому что и свойство `instrument`, и компонент `saxophone` имеют тип `Instrument`.

Однако автоматическое связывание по типу имеет одно ограничение. Представьте, что произойдет, если в конфигурации будет описано более одного компонента, тип которого совпадает с типом автоматически связываемого свойства. В этом случае фреймворк не будет строить предположений о том, какой компонент выбрать для внедрения, а просто вызовет исключение. Следовательно, в конфигурации должен быть только один компонент, тип которого совпадает с типом автоматически связываемого свойства. В конкурсе «Spring Idol» наверняка будет несколько компонентов, типы которых являются подклассами класса `Instrument`.

Чтобы устранить неоднозначность при использовании автоматического связывания по типу, Spring предлагает два решения: либо определить основного кандидата для автоматического связывания, либо исключить компоненты из списка кандидатов на автоматическое связывание.

Чтобы определить основного кандидата для автоматического связывания, в элемент `<bean>` определения компонента необходимо добавить атрибут `primary`. Если механизм автоматического связывания обнаружит только один подходящий компонент с атрибутом `primary`, установленным в значение `true`, он отдаст предпочтение этому компоненту.

Но самое неприятное, что атрибут `primary` получает значение `true` по умолчанию. Это означает, что все кандидаты для автоматического связывания по умолчанию являются основными (и потому ни одному из них не может быть отдано предпочтение). То есть при использовании решения на основе атрибута `primary` необходимо присвоить ему значение `false` во всех остальных компонентах. Например, чтобы указать, что компонент `saxophone` не является основным кандидатом для автоматического связывания со свойством типа `Instruments`:

```
<bean id="saxophone"
      class="com.springinaction.springidol.Saxophone"
      primary="false" />
```

Атрибут `primary` удобно использовать только для определения предпочтительного кандидата. При использовании решения, основанного на исключении некоторых компонентов из рассмотрения механизмом автоматического связывания, следует установить их атрибут `autowire-candidate` в значение `false`, как показано ниже:

```
<bean id="saxophone"
      class="com.springinaction.springidol.Saxophone"
      autowire-candidate="false" />
```

Автоматическое связывание через конструктор

Если компонент настроен на внедрение зависимостей через конструктор, можно просто убрать элементы `<constructor-arg>` из его объявления и позволить фреймворку автоматически выбрать аргументы для передачи конструктору из компонентов, имеющихся в контексте Spring.

Например, взгляните на следующее объявление компонента `duke`:

```
<bean id="duke"
      class="com.springinaction.springidol.PoeticJuggler"
      autowire="constructor" />
```

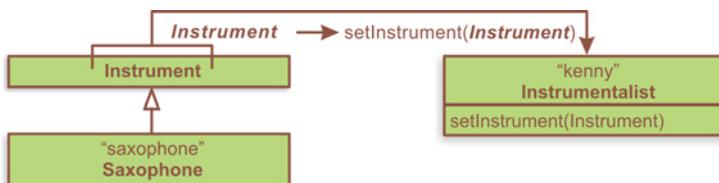


Рис. 4.2. Автоматическое связывание по совпадению типа компонента с типом свойства



Рис. 4.3. При автоматическом связывании через конструктор новый компонент `duke` типа `PoeticJuggler` будет создан вызовом конструктора с аргументом `Poem`

Из этого нового объявления компонента duke исчезли элементы `<constructor-arg>`, а атрибуту `autowire` было присвоено значение `constructor`. Это объявление сообщает фреймворку, что он должен исследовать конструктор класса `PoeticJuggler` и попытаться найти в конфигурации компоненты, соответствующие аргументам одного из конструкторов. В конфигурации уже присутствует определение компонента `sonnet29`, имеющего тип `Poem` и соответствующего аргументу одного из конструкторов класса `PoeticJuggler`. Поэтому для создания компонента duke Spring будет использовать этот конструктор и передаст ему компонент `sonnet29`, как изображено на рис. 4.3.

На автоматическое связывание через конструктор распространяются те же ограничения, что и на автоматическое связывание по типу (`byType`). В случае обнаружения нескольких компонентов, соответствующих аргументам конструктора, фреймврк Spring не будет пытаться угадать, какой из них использовать. Кроме того, если класс имеет несколько конструкторов, каждый из которых отвечает требованиям автоматического связывания, Spring не будет пытаться угадать, какой конструктор использовать.

Автоматическое связывание с автоматическим определением

Если вы желаете организовать автоматическое связывание компонентов, но не можете выбрать наиболее подходящий способ, не волнуйтесь – установив атрибут `autowire` в значение `autodetect`, вы можете позволить контейнеру сделать выбор за вас. Например:

```
<bean id="duke"
      class="com.springinaction.springidol.PoeticJuggler"
      autowire="autodetect" />
```

Когда компонент настроен на автоматическое связывание посредством автоматического определения, фреймврк будет пытаться сначала выполнить автоматическое связывание через конструктор. Если подходящее совпадение не будет найдено, тогда будет предпринята попытка выполнить автоматическое связывание по типу.

Автоматическое связывание по умолчанию

Если вы вдруг обнаружите, что вам придется добавить один и тот же атрибут `autowire` во все компоненты (или в большинство из них), можно просто попросить фреймврк автоматически применить одни и те же настройки автоматического связывания ко всем компонен-

там. Для этого в корневой элемент `<beans>` следует добавить атрибут `default-autowire`:

```
<?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-3.0.xsd"
       default-autowire="byType">
</beans>
```

По умолчанию атрибут `default-autowire` имеет значение `none`, указывая, что настройка автоматического связывания для каждого компонента выполняется индивидуально, посредством атрибута `autowire`. В примере выше ему было присвоено значение `byType`, чтобы указать, что свойства всех компонентов должны автоматически связываться по типу. Но атрибуту `default-autowire` можно присвоить любой другой тип автоматического связывания, который должен применяться ко всем компонентам, описанным в конфигурационном файле.

Обратите внимание: я сказал, что настройка атрибута `default-autowire` будет применяться ко всем компонентам, описанным в конфигурационном файле Spring, но я не сказал, что она будет применяться ко всем компонентам в контексте приложения. Можно создать несколько файлов конфигурации, определяющих параметры одного и того же контекста приложения, и в каждом из них использовать собственные настройки автоматического связывания.

Кроме того, определение настройки автоматического связывания по умолчанию не означает, что она обязательно должна применяться ко всем компонентам. За вами остается возможность переопределить эту настройку в каждом отдельном компоненте, добавив в него атрибут `autowire`.

4.1.2. Смешивание автоматического и явного связывания

Использование механизма автоматического связывания не означает, что нельзя явно связывать некоторые свойства. Вы все еще можете использовать элемент `<property>` для описания любого свойства, как если бы не использовали автоматическое связывание.

Например, чтобы явно связать свойство `instrument` компонента `kenny`, даже при том, что для этого компонента определено использо-

зование автоматического связывания по типу, можно использовать следующее определение:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist"
      autowire="byType">
    <property name="song" value="Jingle Bells" />
    <property name="instrument" ref="saxophone" />
</bean>
```

Как видно из этого примера, смешивание автоматического и явного связывания – отличный способ устраниТЬ неоднозначности, возникающие при автоматическом связывании по типу (*byType*). В конфигурации Spring может быть определено несколько компонентов, реализующих интерфейс *Instrument*. Чтобы предотвратить появление исключения, вызванного неопределенностью выбора из нескольких компонентов типа *Instrument*, можно явно связать свойство *instrument*, фактически исключив его из процедуры автоматического связывания.

Ранее упоминалось, что можно использовать элемент `<null/>`, чтобы принудительно присвоить автоматически связываемому свойству значение `null`. Это особый случай смешивания автоматического и явного связывания. Например, если потребуется, чтобы свойство *instrument* компонента *kenny* получило значение `null`, можно использовать следующее его определение:

```
<bean id="kenny"
      class="com.springinaction.springidol.Instrumentalist"
      autowire="byType">
    <property name="song" value="Jingle Bells" />
    <property name="instrument"><null/></property>
</bean>
```

Конечно, это всего лишь иллюстрация. Если внедрить значение `null` в свойство *instrument*, это приведет к исключению *NullPointerException* в вызове метода *perform()*.

Заключительное замечание о смешанном связывании: при использовании автоматического связывания через конструктор следует дать фреймворку Spring возможность связывать все аргументы конструктора – нельзя использовать элементы `<constructor-arg>` вместе с автоматическим связыванием через конструктор.

4.2. Связывание посредством аннотаций

С выходом версии Spring 2.5 появился один из самых интересных способов связывания компонентов, основанный на автоматическом связывании свойств с использованием аннотаций. Автоматическое связывание с использованием аннотаций мало чем отличается от использования атрибута `autowire` в XML-файле конфигурации. Но он обеспечивает возможность более точного управления автоматическим связыванием, позволяя выборочно объявлять свойства, доступные для автоматического связывания.

Поддержка автоматического связывания посредством аннотаций по умолчанию отключена. Поэтому, прежде чем использовать этот способ, необходимо включить его поддержку в конфигурационном файле. Проще всего это сделать с помощью элемента `<context:annotation-config>` из пространства имен `context`:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:annotation-config />

    <!-- здесь находятся объявления компонентов -->

</beans>
```

Элемент `<context:annotation-config>` сообщает фреймворку Spring, что для связывания свойств должен использоваться механизм на основе аннотаций. Добавив этот элемент, можно приступить к добавлению соответствующих аннотаций в программный код, чтобы определить свойства, методы и конструкторы для автоматического связывания.

В Spring 3 поддерживаются несколько аннотаций для автоматического связывания:

- ❑ аннотация `@Autowired`, определяемая самим фреймворком Spring;
- ❑ аннотация `@Inject` из JSR-330;
- ❑ аннотация `@Resource` из JSR-250.

Сначала мы познакомимся с особенностями использования аннотации `@Autowired`, а затем попробуем внедрить зависимости с использованием аннотаций `@Inject` и `@Resource`, определяемых стандартами JSR-330 и JSR-250 соответственно.

4.2.1. Использование аннотации `@Autowired`

Предположим, что необходимо с помощью аннотации `@Autowired` обеспечить автоматическое внедрение значения в свойство `instrument` компонента `Instrumentalist`. Для этого можно аннотировать метод `setInstrument()`, как показано ниже:

```
@Autowired
public void setInstrument(Instrument instrument) {
    this.instrument = instrument;
}
```

Теперь можно убрать элемент `<property>`, связывающий свойство `instrument` компонента типа `Instrumentalist`. Когда фреймворк обнаружит, что метод `setInstrument()` снабжен аннотацией `@Autowired`, он попытается выполнить автоматическое связывание по типу (`byType`).

Самая интересная особенность аннотации `@Autowired` состоит в том, что ее необязательно применять к методу записи. Эту аннотацию можно применить к любому методу, чтобы обеспечить автоматическое внедрение ссылки на компонент:

```
@Autowired
public void heresYourInstrument(Instrument instrument) {
    this.instrument = instrument;
}
```

Аннотацию `@Autowired` можно даже применить к конструктору:

```
@Autowired
public Instrumentalist(Instrument instrument) {
    this.instrument = instrument;
}
```

При использовании с конструктором аннотация `@Autowired` указывает, что механизм автоматического связывания должен использовать конструктор при создании компонента, даже если в описании

компонент в конфигурационном XML-файле отсутствует элемент <constructor-arg>.

Кроме того, аннотировать можно свойства непосредственно и вообще избавиться от методов записи:

```
@Autowired  
private Instrument instrument;
```

Как видите, ключевое слово `private` не является препятствием для аннотации `@Autowired`. Несмотря на то что свойство `instrument` объявлено частным, оно будет доступно для автоматического связывания. Неужели для аннотации `@Autowired` нет никаких препятствий?

В действительности существуют несколько обстоятельств, мешающих работе аннотации `@Autowired`. В частности, должен существовать точно один компонент, подходящий для связывания со свойством или параметром, аннотированным с помощью `@Autowired`. Если имеется несколько подходящих компонентов или нет ни одного, аннотация `@Autowired` столкнется с определенными проблемами.

К счастью, имеется возможность помочь аннотации `@Autowired` в подобных ситуациях. Сначала посмотрим, как помочь аннотации `@Autowired` в случае, когда нет ни одного подходящего компонента.

Необязательное автоматическое связывание

По умолчанию аннотация `@Autowired` строго требует, чтобы аннотируемые ею свойства и параметры были связаны. Если свойство, отмеченное аннотацией `@Autowired`, не может быть связано с каким-либо компонентом, операция автоматического связывания терпит неудачу (с возбуждением неприятного исключения `NoSuchBeanDefinitionException`). Такое поведение, когда вместо исключения `NullPointerException` во время выполнения фреймворк терпит неудачу при попытке выполнить автоматическое связывание, вполне может быть желаемым.

Но точно так же может случиться, что свойство не обязательно должно быть связано и значение `null` в нем является вполне допустимым. В этой ситуации можно определить необязательное автоматическое связывание, установив атрибут `required` аннотации `@Autowired` в значение `false`. Например:

```
@Autowired(required=false)  
private Instrument instrument;
```

Здесь фреймворк Spring попытается связать свойство `instrument`, но если он не найдет подходящего компонента типа `Instrument`, ничего страшного не произойдет. Свойство просто получит значение `null`.

Примечательно, что атрибут `required` можно использовать везде, где допускается использовать аннотацию `@Autowired`. Но когда эта аннотация применяется к конструкторам, только один из них может быть отмечен аннотацией `@Autowired` с атрибутом `required`, установленным в значение `true`. Для всех остальных конструкторов аннотации `@Autowired` должны иметь атрибут `required` со значением `false`. Кроме того, при наличии нескольких конструкторов, отмеченных аннотацией `@Autowired`, Spring будет выбирать конструктор, где больше всего аргументов могут быть связаны.

Уточнение неоднозначных зависимостей

С другой стороны, проблема может быть обусловлена не отсутствием компонентов, а неоднозначностью выбора. Может так случиться, что компонентов будет слишком много (по крайней мере, два), каждый из которых одинаково пригоден для внедрения в свойство или параметр.

Например, допустим, что имеются два компонента, реализующих интерфейс `Instrument`. В этом случае аннотация `@Autowired` не сможет определить, какой из них действительно требуется внедрить. Поэтому, вместо того чтобы пытаться угадать, фреймворк возбудит исключение `NoSuchBeanDefinitionException`.

Чтобы помочь аннотации `@Autowired` выбрать требуемый компонент, можно добавить аннотацию `@Qualifier`.

Например, чтобы гарантировать, что Spring выберет гитару для компонента `eddie`, даже если существуют другие компоненты, которые могут быть связаны с его свойством `instrument`, можно добавить аннотацию `@Qualifier`, ссылающуюся на компонент с именем `guitar`:

```
@Autowired  
@Qualifier("guitar")  
private Instrument instrument;
```

Как показано в этом примере, аннотация `@Qualifier` определяет, что для связывания должен использоваться компонент с идентификатором `guitar`.

На первый взгляд может сложиться впечатление, что аннотация `@Qualifier` выполняет переход от автоматического связывания по типу к автоматическому связыванию по имени. В данном приме-

ре именно это и происходит. Но в действительности аннотация `@Qualifier` лишь сужает выбор компонентов, доступных для автоматического связывания. Просто так получается, что ссылка на идентификатор компонента является одним из способов сузить выбор до единственного компонента.

Однако сузить выбор можно, не только указав идентификатор компонента. Имеется также возможность явно указать требуемый компонент. Например, предположим, что компонент `guitar` определен в XML-файле, как показано ниже:

```
<bean class="com.springinaction.springidol.Guitar">
    <qualifier value="stringed" />
</bean>
```

Здесь элемент `<qualifier>` квалифицирует компонент `guitar` как струнный (`stringed`) инструмент. Вместо квалификации компонента в XML-файле аннотацией `@Qualifier` можно также аннотировать сам класс `Guitar`:

```
@Qualifier("stringed")
public class Guitar implements Instrument {
    ...
}
```

Уточнение выбора компонентов с помощью строковых идентификаторов, будь то значение атрибута `id` компонента или какой-то другой квалификатор, реализуется довольно просто. Однако квалификация может принимать гораздо более сложные формы. Фактически можно даже создать свои собственные аннотации, уточняющие выбор.

Создание собственных квалификаторов

Чтобы создать собственную аннотацию, уточняющую выбор, достаточно просто определить аннотацию, которая сама отмечена аннотацией `@Qualifier`. Например, создадим аннотацию `@StringedInstrument`, которая будет играть роль квалификатора. Определение этой аннотации приводится в листинге 4.1.

Листинг 4.1. Использование аннотации `@Qualifier` для определения собственной аннотации-квалификатора

```
package com.springinaction.springidol.qualifiers;
import java.lang.annotation.ElementType;
```

```
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import org.springframework.beans.factory.annotation.Qualifier;

@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface StringedInstrument {
```

После определения аннотации `@StringedInstrument` ее можно использовать для аннотирования класса `Guitar` вместо аннотации `@Qualifier`:

```
@StringedInstrument
public class Guitar implements Instrument {
    ...
}
```

А затем пометить аннотацией `@StringedInstrument` автоматически связываемое свойство `instrument`:

```
@Autowired
@StringedInstrument
private Instrument instrument;
```

Когда фреймворк попытается выполнить автоматическое связывание свойства `instrument`, его выбор сузится от всех компонентов типа `Instrument` до тех из них, что отмечены аннотацией `@StringedInstrument`. Пока в приложении остается единственный компонент с аннотацией `@StringedInstrument`, он будет внедрен в свойство `instrument`.

Когда появятся несколько компонентов с аннотацией `@StringedInstrument`, потребуется добавить дополнительное уточнение, которое сузит выбор до одного компонента. Например, представьте, что помимо компонента `Guitar` имеется еще компонент `HammeredDulcimer`, так же отмеченный аннотацией `@StringedInstrument`. Единственное ключевое отличие между гитарой и цимбалами состоит в том, что на цимбалах играют маленькими деревянными палочками (или моточками).

Поэтому, чтобы обеспечить дополнительную квалификацию класса `Guitar`, можно определить еще одну аннотацию с именем `@Strummed`:

```
@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface Strummed {
}
```

Теперь аннотацию `@Strummed` можно добавить к свойству `instrument`, чтобы ограничить выбор щипковыми струнными инструментами:

```
@Autowired
@StringedInstrument
@Strummed
private Instrument instrument;
```

Если класс `Guitar` окажется единственным с аннотациями `@Strummed` и `@StringedInstrument`, тогда в свойство `instrument` будет внедрен компонент этого класса.

Дальше можно было бы обсудить возможность добавления таких компонентов, как `Ukelele` (гавайская гитара) или `Mandolin` (мандолина), но мы не можем продолжать это до бесконечности. Просто помните, что в этом случае потребуется дальнейшее уточнение выбора из этих дополнительных щипковых струнных инструментов.

Применение аннотации `@Autowired` – это лишь один из способов, способствующих уменьшению размеров конфигурационных XML-файлов. Но он создает зависимость от фреймворка Spring (несмотря на то что эта зависимость является всего лишь аннотацией). К счастью, Spring поддерживает также стандартные альтернативы аннотации `@Autowired`. Посмотрим далее, как использовать аннотацию `@Inject`, определяемую спецификацией «Dependency Injection for Java».

4.2.2. Автоматическое связывание с применением стандартной аннотации `@Inject`

С целью унификации модели программирования для применения в различных фреймворках, реализующих внедрение зависимостей, организация Java Community Process недавно опубликовала спецификацию «Dependency Injection for Java». Известная в организации Java Community Process под названием JSR-330, эта спецификация описывает обобщенную модель внедрения зависимостей в языке

Java. Эта модель поддерживается фреймворком Spring, начиная с версии Spring 3.0¹.

Центральное положение в модели JSR-330 занимает аннотация `@Inject`. Эта аннотация является практически полным аналогом аннотации `@Autowired` фреймворка Spring. Поэтому вместо аннотации `@Autowired` для описания свойства `instrument` можно использовать аннотацию `@Inject`:

```
@Inject  
private Instrument instrument;
```

Как и `@Autowired`, аннотацию `@Inject` можно использовать для автоматического связывания свойств, методов и конструкторов. В отличие от `@Autowired`, аннотация `@Inject` не имеет атрибута `required`. Поэтому неудовлетворенные зависимости, описываемые аннотацией `@Inject`, будут приводить к исключению.

В дополнение к аннотации `@Inject` в модели JSR-330 предусмотрена еще одна хитрость. Вместо того чтобы внедрять зависимости непосредственно, от аннотации `@Inject` можно потребовать, чтобы она внедряла ссылку на реализацию интерфейса `Provider`. Среди всего прочего интерфейс `Provider` обеспечивает возможность отложенного внедрения ссылок на компоненты и внедрение нескольких экземпляров компонента.

Например, представьте, что имеется класс `KnifeJuggler` (жонглер ножами), в экземпляр которого должен быть внедрен один или более экземпляров класса `Knife` (нож). Предположим, что компонент типа `Knife` объявлен как имеющий область действия `prototype`, тогда следующий конструктор `KnifeJuggler` сможет получить пять компонентов типа `Knife`:

```
private Set<Knife> knives;  
  
@Inject  
public KnifeJuggler(Provider<Knife> knifeProvider) {  
    knives = new HashSet<Knife>();  
    for (int i = 0; i < 5; i++) {  
        knives.add(knifeProvider.get());  
    }  
}
```

¹ Фреймворк Spring – не единственный, поддерживающий JSR-330. Модель JSR-330 также поддерживается фреймворками Google Guice и Picocontainer.

Вместо экземпляров класса `Knife` на этапе конструирования экземпляр `KnifeJuggler` получит экземпляр `Provider<Knife>`. На этом этапе будет внедрен только экземпляр объекта-посредника `Provider`. Внедрение фактических объектов типа `Knife` будет отложено до вызова метода `get()` объекта-посредника. В данном случае метод `get()` вызывается пять раз. А поскольку компонент типа `Knife` имеет область действия `prototype`, во множестве `knives` будет сохранено пять различных объектов типа `Knife`.

Уточнение связываемых свойств

Как видите, аннотации `@Inject` и `@Autowired` имеют много общего. И подобно `@Autowired`, аннотация `@Inject` также подвержена проблеме неоднозначности выбора компонента для внедрения. Аннотация `@Inject` тоже может сопровождаться уточняющей аннотацией `@Named`, подобной аннотации `@Qualifier`.

Аннотация `@Named` действует подобно аннотации `@Qualifier`, как показано ниже:

```
@Inject  
@Named("guitar")  
private Instrument instrument;
```

Важным отличием аннотации `@Qualifier`, определяемой фреймворком Spring, от аннотации `@Named`, определяемой спецификацией JSR-330, является семантика их использования. Аннотация `@Qualifier` помогает сузить круг компонентов, доступных для выбора (используя идентификатор компонента как один из возможных квалифиликаторов), тогда как аннотация `@Named` однозначно ссылается на идентификаторы компонентов.

Создание собственных квалификаторов JSR-330

Как оказывается, спецификация JSR-330 определяет собственную аннотацию `@Qualifier` в пакете `javax.inject`. В отличие от аннотации `@Qualifier`, определяемой фреймворком Spring, версия JSR-330 не имеет самостоятельного значения. Она предназначена для определения пользовательских аннотаций-квалификаторов, почти так же, как аннотация `@Qualifier`, определяемая фреймворком Spring¹.

¹ В действительности аннотация `@Named` сама определена посредством аннотации `@Qualifier`.

Например, следующий листинг демонстрирует определение новой аннотации `@StringedInstrument`, созданной с помощью аннотации `@Qualifier`, определяемой спецификацией JSR-330.

Листинг 4.2. Создание собственного квалификатора с помощью аннотации `@Qualifier`, определяемой спецификацией JSR-330

```
package com.springinaction.springidol;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

import javax.inject.Qualifier;

@Target({ElementType.FIELD, ElementType.PARAMETER, ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Qualifier
public @interface StringedInstrument { }
```

Как видите, единственное отличие между листингами 4.2 и 4.1 заключается в инструкции импортирования определения аннотации `@Qualifier`. В листинге 4.1 использовалось определение из пакета `org.springframework.beans.factory.annotation`. А в этом примере была задействовано определение стандартной аннотации `@Qualifier` из пакета `javax.inject`. Во всем остальном примеры совершенно идентичны.

Автоматическое связывание на основе аннотаций отлично подходит для внедрения ссылок на компоненты и избавления от лишних элементов `<property>` в XML-файле конфигурации Spring. Но можно ли использовать аннотации для внедрения строковых и других простых значений в свойства?

4.2.3. Использование выражений в аннотациях внедрения зависимостей

Помимо внедрения ссылок на компоненты с помощью аннотаций автоматического связывания, может появиться желание использовать аннотации для внедрения простых значений. В версии Spring 3.0 появилась новая аннотация `@Value`, позволяющая внедрять простые значения таких типов, как `int`, `boolean` и `String`.

Аннотация @Value проста в использовании, но, как будет показано чуть ниже, она обладает широкими возможностями. Ее применение заключается в том, чтобы добавить аннотацию @Value перед свойством, методом или параметром метода и передать ей строку с выражением, результат которого должен быть внедрен. Например:

```
@Value("Eruption")
private String song;
```

Здесь в строковое свойство внедряется обычное строковое значение. Но строковый параметр аннотации @Value в действительности является выражением, которое может возвращать значение любого типа, благодаря чему аннотация @Value может применяться к свойствам любых типов.

Внедрение жестко определенных значений с использованием аннотации @Value само по себе малоинтересно. Если заранее известные значения можно присваивать непосредственно в программном коде Java, почему бы тогда не отказаться от аннотации @Value и не присвоить требуемое значение свойству напрямую? В данном примере аннотация @Value выглядит как излишество.

Как оказывается, внедрение простых значений не является коньком аннотации @Value. Полная ее мощь заключается в возможности использовать выражения на языке SpEL. Напомню, что язык SpEL позволяет динамически вычислять сложные выражения во время выполнения и использовать полученные значения для внедрения в свойства компонентов. Это делает аннотацию @Value мощным инструментом внедрения.

Например, вместо внедрения жестко определенного значения в свойство song попробуем задействовать выражение на языке SpEL, чтобы внедрить значение системного свойства:

```
@Value("#{systemProperties.myFavoriteSong}")
private String song;
```

В этом примере аннотация @Value показала свои возможности. Это не только инструмент доставки статических значений, это еще и эффективный способ внедрения значений динамически вычисляемых выражений на языке SpEL.

Как видите, автоматическое связывание – мощный прием. Если позволить фреймворку Spring автоматически связывать компонен-

ты, это поможет уменьшить размеры XML-файлов конфигурации приложения. Более того, автоматическое связывание позволяет еще больше ослабить связанность компонентов, отделяя объявления компонентов друг от друга.

А теперь познакомимся с механизмом автоматического определения компонентов, чтобы увидеть, как фреймворк Spring может не только связывать компоненты, но и автоматически определять, какие компоненты должны быть зарегистрированы в контексте Spring.

4.3. Автоматическое определение компонентов

Когда в конфигурацию Spring добавляется элемент `<context:annotation-config>`, он сообщает фреймворку о необходимости учитывать аннотации в компонентах при их связывании. Даже при том, что применение элемента `<context:annotation-config>` способно помочь избавиться от большинства элементов `<property>` и `<constructor-arg>` в файле конфигурации, это не избавляет от необходимости объявлять компоненты с помощью элемента `<bean>`.

Но у фреймворка Spring есть в запасе еще одна хитрость. Элемент `<context:component-scan>` делает все то же, что и элемент `<context:annotation-config>`, плюс он настраивает фреймворк на автоматическое определение компонентов и их объявление. Это означает, что большинство (если не все) компонентов в приложении на основе Spring можно объявить и связать без использования элемента `<bean>`.

Чтобы включить режим автоматического определения компонентов, необходимо вместо элемента `<context:annotation-config>` добавить элемент `<context:component-scan>`:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <context:component-scan
        base-package="com.springinaction.springidol">
    </context:component-scan>
</beans>
```

Элемент `<context:component-scan>` заставляет фреймворк просмотреть пакет и все вложенные в него пакеты package и отыскать классы, которые можно автоматически зарегистрировать в виде компонентов в контейнере Spring. Атрибут `base-package` элемента `<context:component-scan>` определяет пакет, откуда следует начинать поиск.

Как элемент `<context:component-scan>` определяет, какие классы регистрировать в виде компонентов Spring?

4.3.1. Аннотирование компонентов для автоматического определения

По умолчанию элемент `<context:component-scan>` требует выполнить поиск классов, отмеченных одной из нескольких специальных аннотаций:

- `@Component` – универсальная аннотация, указывающая, что класс является компонентом Spring;
- `@Controller` – указывает, что класс определяет контроллер Spring MVC;
- `@Repository` – указывает, что класс определяет репозиторий данных;
- `@Service` – указывает, что класс определяет службу;
- любая пользовательская аннотация, определенная с помощью аннотации `@Component`.

Для примера предположим, что контекст нашего приложения содержит только компоненты `eddie` и `guitar`. В этом случае из XML-файла конфигурации можно удалить явные объявления `<bean>`, добавив в него элемент `<context:component-scan>` и пометив классы `Instrumentalist` и `Guitar` аннотацией `@Component`.

Сначала добавим аннотацию `@Component` в класс `Guitar`:

```
package com.springinaction.springidol;

import org.springframework.stereotype.Component;

@Component
public class Guitar implements Instrument {
    public void play() {
        System.out.println("Strum strum strum");
    }
}
```

Когда фреймворк будет просматривать пакет com.springinaction.springidol, он обнаружит, что класс Guitar отмечен аннотацией @Component, и автоматически зарегистрирует его как компонент. По умолчанию идентификатор компонента генерируется из имени класса, где первый символ имени замещается этим же символом в нижнем регистре. В случае с классом Guitar компонент получит идентификатор guitar.

Теперь аннотируем класс Instrumentalist:

```
package com.springinaction.springidol;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component("eddie")
public class Instrumentalist implements Performer {
    // ...
}
```

В данном случае желаемый идентификатор компонента был передан аннотации @Component в виде параметра. По умолчанию компонент получил бы идентификатор instrumentalist, но, чтобы сохранить непротиворечивость с предыдущими примерами, для него явно был определен идентификатор eddie.

Автоматическое определение на основе аннотаций – это лишь одна из возможностей элемента <context:component-scan>. Посмотрим далее, как настроить элемент <context:component-scan>, чтобы он выполнял поиск кандидатов в компоненты по другим признакам.

4.3.2. Включение фильтров в элемент component-scans

Как оказывается, элемент <context:component-scan> обеспечивает значительную гибкость в отношении поиска кандидатов в компоненты. Дополнительные настройки процедуры поиска можно выполнить, добавляя элементы <context:include-filter> и <context:excludefilter> в <context:component-scan>.

Для демонстрации действия фильтров в элементе component-scan посмотрим, что можно предпринять, чтобы обеспечить автоматическую регистрацию всех классов, реализующих интерфейс Instrument, используя стратегию на основе аннотаций. Для этого необходимо открыть исходные тексты всех реализаций интерфейса Instrument и

пометить их аннотацией @Component (или любой другой аннотации из представленных выше). Это как минимум неудобно. А если в приложении используется сторонняя реализация Instrument, это вообще может оказаться невозможным.

Поэтому вместо использования аннотаций потребуем от элемента <context:component-scan> автоматически зарегистрировать все классы, экземпляры которых могут присваиваться свойствам типа Instrument, добавив фильтр включения, как показано ниже:

```
<context:component-scan
    base-package="com.springinaction.springidol">
    <context:include-filter type="assignable"
        expression="com.springinaction.springidol.Instrument"/>
</context:component-scan>
```

Атрибуты type и expression элемента <context:include-filter> в совокупности определяют стратегию поиска. В данном случае мы потребовали зарегистрировать в виде компонентов Spring все классы, экземпляры которых могут присваиваться свойствам типа Instrument. Однако существуют и другие типы фильтров, которые перечислены в табл. 4.1.

Таблица 4.1. Поиск компонентов можно настраивать, используя любой из пяти типов фильтров

Тип фильтра	Описание
annotation	Отыскивает классы, отмеченные указанной аннотацией на уровне типа. Аннотация определяется атрибутом expression
assignable	Отыскивает классы, экземпляры которого могут присваиваться свойствам указанного типа. Тип свойств определяется атрибутом expression
aspectj	Отыскивает классы, тип которых соответствует выражению типа AspectJ, указанному в атрибуте expression
custom	Использует пользовательскую реализацию org.springframework.core.type.TypeFilter, указанную в атрибуте expression
regex	Отыскивает классы, имена которых соответствуют регулярному выражению, указанному в атрибуте expression

По аналогии с элементом <context:include-filter>, определяющим классы, которые должны регистрироваться в виде компонентов, в элементе <context:component-scan> можно также использовать элемент <context:exclude-filter>, описывающий классы, которые не должны регистрироваться. Например, чтобы зарегистрировать все реали-



зации интерфейса `Instrument`, кроме отмеченных пользовательской аннотацией `@SkipIt`:

```
<context:component-scan
    base-package="com.springinaction.springidol">
    <context:include-filter type="assignable"
        expression="com.springinaction.springidol.Instrument"/>
    <context:exclude-filter type="annotation"
        expression="com.springinaction.springidol.SkipIt"/>
</context:component-scan>
```

Когда дело доходит до организации фильтрации в элементе `<context:component-scan>`, его возможности становятся практически безграничны. Тем не менее чаще всего используется стратегия по умолчанию, основанная на применении аннотаций. И именно она чаще всего будет использоваться в этой книге.

4.4. Конфигурирование Spring в программном коде на Java

Хотите верьте, хотите нет, но не все разработчики являются ярыми поклонниками XML. В действительности некоторые из них являются членами клуба Настоящих Мужчин, Ненавистников XML. Они очень хотели бы избавить мир от угловых скобок. Длинная история использования XML в Spring приучила некоторых противников XML использовать его.

Тем, кто принадлежит числу противников XML, версия Spring 3.0 может предложить кое-что особенное. Теперь есть возможность конфигурировать приложения на основе Spring практически без XML, исключительно в программном коде на языке Java. И даже у тех, кто не является противником XML, может возникнуть желание попробовать выполнить конфигурирование на языке Java, потому что, как будет показано чуть ниже, этот прием обладает некоторыми возможностями, недоступными в XML.

4.4.1. Подготовка к конфигурированию на языке Java

Несмотря на то что прием конфигурирования на языке Java позволяет описать конфигурацию приложения практически без использования XML, тем не менее некоторый объем XML-кода все

же необходим, чтобы подготовить к использованию конфигурацию на языке Java:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <context:component-scan
        base-package="com.springinaction.springidol" />
</beans>
```

Выше уже рассказывалось, что элемент `<context:component-scan>` обеспечивает автоматическую регистрацию компонентов, отмеченных специальными аннотациями. Но он также автоматически загружает Java-классы, реализующие конфигурирование, отмеченные аннотацией `@Configuration`. В данном случае атрибут `base-package` сообщает фреймворку Spring, что классы, отмеченные аннотацией `@Configuration`, следует искать в пакете `com.springinaction.springidol`.

4.4.2. Определение конфигурационных классов

Когда мы начинали знакомиться с конфигурациями Spring в формате XML, я показал фрагмент с элементом `<beans>` из пространства имен `beans`, играющим роль корневого элемента. Его эквивалентом на языке Java является *класс*, отмеченный аннотацией `@Configuration`. Например:

```
package com.springinaction.springidol;
import org.springframework.context.annotation.Configuration;

@Configuration
public class SpringIdolConfig {
    // Здесь находятся методы, определяющие компоненты
}
```

Аннотация `@Configuration` подсказывает фреймворку Spring, что данный класс содержит одно или более определений компонен-



тов. Объявления компонентов – это обычные методы, отмеченные аннотацией @Bean. Посмотрим, как использовать аннотацию @Bean для связывания компонентов в описании конфигурации на языке Java.

4.4.3. Объявление простого компонента

Для объявления компонента типа Juggler с идентификатором duke в главе 2 мы использовали элемент <bean>. В конфигурации на языке Java приложения Spring Idol компонент duke можно определить как метод с аннотацией @Bean:

```
@Bean  
public Performer duke() {  
    return new Juggler();  
}
```

Этот простой метод в конфигурации на языке Java является эквивалентом элемента <bean>, созданного ранее. Аннотация @Bean сообщает фреймворку Spring, что данный метод вернет объект, который должен быть зарегистрирован в контексте приложения Spring как компонент. Компонент получит идентификатор, совпадающий с именем метода. Все операции, выполняемые внутри метода, в конечном итоге должны создавать компонент.

В данном случае объявление компонента выглядит очень просто. Метод просто создает и возвращает экземпляр класса Juggler. Этот объект будет зарегистрирован фреймворком Spring в контексте приложения с идентификатором duke.

Хотя этот метод объявления компонента в значительной степени является эквивалентом объявления в формате XML, тем не менее он иллюстрирует одно из самых больших преимуществ реализации конфигурации на языке Java перед оформлением конфигурации в формате XML. В XML-версии оба значения, тип и идентификатор компонента, определяются строковыми атрибутами. Недостатком строковых идентификаторов является невозможность их проверки на этапе компиляции. Можно переименовать класс Juggler в программном коде и забыть внести соответствующие изменения в конфигурацию в формате XML.

В конфигурации на языке Java не используются строковые атрибуты. Идентификатор компонента и его тип являются частью сигнатуры метода. Фактическое создание компонента выполняется

в теле метода. Поскольку в данном случае конфигурация описывается программным кодом, появляется дополнительное преимущество, обусловленное дополнительными проверками, выполняемыми на этапе компиляции и гарантирующими, что компонент будет иметь действительный тип, а его идентификатор будет уникальным.

4.4.4. Внедрение зависимостей в конфигурации на языке Java

Если объявление компонента в конфигурации на языке Java – не более чем метод, возвращающий экземпляр класса, тогда как в этом случае реализовать внедрение зависимостей? Фактически, если следовать идиомам программирования на языке Java, внедрение реализуется очень просто.

Рассмотрим сначала, как реализовать внедрение значений в компонент. Ранее было показано, как с помощью элемента `<constructor-arg>` в конфигурации в формате XML создать жонглера (компонент типа `Juggler`), жонглирующего 15 мячиками. В конфигурации на языке Java достаточно просто передать требуемое число конструктору:

```
@Bean
public Performer duke15() {
    return new Juggler(15);
}
```

Как видите, конфигурация на языке Java выглядит вполне естественно и позволяет определять компоненты любыми доступными способами. Внедрение через метод записи на языке Java выглядит не менее естественно:

```
@Bean
public Performer kenny() {
    Instrumentalist kenny = new Instrumentalist();
    kenny.setSong("Jingle Bells");
    return kenny;
}
```

Внедрение простых значений реализуется очень просто. А как обстоит дело со ссылками на другие компоненты? Все так же просто.

Для иллюстрации сначала объявим компонент sonnet29:

```
@Bean  
private Poem sonnet29() {  
    return new Sonnet29();  
}
```

Это объявление еще одного простого компонента на языке Java, мало чем отличающееся от объявления компонента duke. Теперь создадим компонент типа PoeticJuggler и внедрим в него компонент sonnet29 через конструктор:

```
@Bean  
public Performer poeticDuke() {  
    return new PoeticJuggler(sonnet29());  
}
```

Внедрение другого компонента заключается в использовании ссылки на метод определения внедряемого компонента. Но эта простота кажущаяся. В действительности все немного сложнее.

В конфигурации Spring на языке Java ссылка на компонент через метод с его объявлением – это не то же самое, что вызов метода. Если бы это было так, то всякий раз, вызывая метод sonnet29(), мы получали бы новый экземпляр компонента. Фреймворк Spring действует немного тоньше.

Пометив метод sonnet29() аннотацией @Bean, мы сообщаем фреймворку, что этот метод определяет компонент для регистрации в контексте приложения. Поэтому при каждом обращении к этому методу внутри другого метода объявления компонента Spring будет перехватывать вызов метода и пытаться отыскать компонент в контексте приложения, не позволяя этому методу создать новый экземпляр.

4.5. В заключение

За годы своего существования фреймворк Spring не раз подвергался критике за слишком обширное использование формата XML. Несмотря на значительные успехи Spring в направлении упрощения разработки корпоративных приложений на языке Java, многие разработчики предпочитают не видеть все эти угловые скобки.

В ответ на критику Spring предлагает несколько способов уменьшения размеров конфигурации приложений в формате XML, вплоть до

полного ее удаления. В этой главе мы увидели, как с помощью механизма автоматического связывания можно избавиться от элементов `<property>` и `<constructor-arg>`. Благодаря элементу `<context:component-scan>` можно обеспечить автоматическую настройку всех компонентов. Мы также увидели, как полную конфигурацию приложения на основе Spring можно реализовать на языке Java, полностью избавившись от XML.

К настоящему моменту мы познакомились с несколькими способами объявления компонентов и внедрения их зависимостей. В следующей главе будет рассматриваться поддержка в Spring аспектно-ориентированного программирования и ее использование для добавления в компоненты новых функциональных возможностей, хотя и важных для работы приложения, но не являющихся основными для самих компонентов.



Глава 5. Аспектно-ориентированный Spring

В этой главе рассматриваются следующие темы:

- ❑ основы аспектно-ориентированного программирования;
- ❑ создание аспектов из POJO;
- ❑ использование аннотаций @AspectJ;
- ❑ внедрение зависимостей в аспекты AspectJ.

Когда я писал эту главу, в Техасе (где я живу) несколько дней стояла рекордная жара. Было очень жарко. При такой погоде кондиционеры оказываются просто незаменимы. Но беда в том, что кондиционеры потребляют электроэнергию, а электроэнергия стоит денег. У нас не так много возможностей, чтобы не платить за прохладный и удобный дом. Это обусловлено тем, что в каждом доме имеется электросчетчик, измеряющий каждый киловатт, и раз в месяц кто-нибудь приезжает для снятия показаний электросчетчика, чтобы электрическая компания знала, какую сумму выставить нам в счете.

Теперь представьте, что произойдет, если счетчики исчезнут и никто не будет приезжать снимать показания. Предположим, что на плечи домовладельца будет возложена обязанность вести учет объема потребления электроэнергии и сообщать его электрической компании. Я допускаю, что найдутся особенно педантичные домовладельцы, которые будут скрупулезно учитывать, сколько было потрачено электроэнергии на свет, телевизор и кондиционер, но большинство не будут утруждать себя этим. Слишком хлопотно контролировать потребление электроэнергии вручную и слишком облазнительно платить за нее поменьше.

Возможно, для потребителя учет электроэнергии «под честное слово» может выглядеть и неплохо, но для электрических компаний такое положение дел далеко от идеала. Именно поэтому в наших домах стоят электросчетчики и раз в месяц к нам приходят счетчики, чтобы снять показания.

Некоторые функции в программных системах напоминают электросчетчики в наших домах. Они должны применяться в нескольких точках в пределах приложения, но не желательно вызывать их явно в каждой точке.

Учет потребления электроэнергии является важной функцией, но не первоочередной в понимании большинства. Скосить траву на лужайке, пропылесосить ковер и почистить ванную комнату – вот важные дела для домовладельца. Учет потребления электроэнергии в их домах – это пассивное занятие с точки зрения домовладельца. (Хотя было бы здорово, если бы уход за лужайкой тоже превратился в пассивное занятие, особенно в такую жару.)

Некоторые операции, выполняемые программами, являются общими для большинства приложений. Регистрация событий, обеспечение безопасности и управление транзакциями – все это важные операции, но должны ли прикладные объекты принимать активное участие в них? Или для прикладных объектов лучше сосредоточиться на решении проблем предметной области, а управление некоторыми аспектами предоставить кому-то другому?

Функции, охватывающие несколько точек приложения, в разработке программного обеспечения называются *сквозными*. Как правило, сквозные функции концептуально отделены (но часто встроены) от основной логики приложения. Отделение основной логики от сквозных функций – это именно то, для чего предназначено *аспектно-ориентированное программирование* (AOP).

В главе 2 рассказывалось, как использовать прием внедрения зависимостей (DI) для настройки и управления прикладными объектами. Подобно тому, как внедрение зависимостей помогает отделить прикладные объекты друг от друга, аспектно-ориентированное программирование помогает отделить сквозные функции от объектов, на которые они влияют.

Реализация регистрации событий является типичным примером применения аспектов, но это далеко не единственная область, где они могут пригодиться. Далее в книге будет представлено несколько практических примеров применения аспектов, включая декларативные транзакции, обеспечение безопасности и кеширование.

В этой главе исследуется поддержка аспектов в Spring, включая объявление аспектами обычных классов и особенности применения аннотаций для создания аспектов. Кроме того, здесь будет представлено расширение AspectJ, еще одна популярная реализация AOP, способная дополнить реализацию AOP в Spring. Но сначала,

прежде чем погрузиться в проблемы управления транзакциями, безопасности и кеширования, посмотрим, как реализуются аспекты в фреймворке Spring, начав с нескольких примеров, демонстрирующих основы АОР.

5.1. Знакомство с АОР

Как отмечалось ранее, аспекты помогают отделить сквозные функции. Проще говоря, сквозные функции могут быть описаны как некоторая функциональность, затрагивающая множество мест в приложении. Обеспечение безопасности, например, является такой сквозной функцией – правила соблюдения безопасности могут затрагивать множество методов в приложении. Рисунок 5.1 дает визуальное представление сквозных функций.

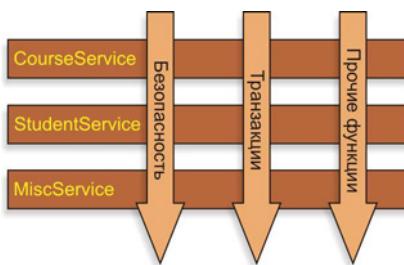


Рис. 5.1. Аспекты отделяют сквозные функции, реализуя логику, охватывающую множество объектов в приложении

Рисунок 5.1 представляет типичное приложение, разбитое на модули. Главная задача каждого из них – предоставление услуг в конкретной предметной области. Но каждый из этих модулей также требует выполнения вспомогательных функций, таких как обеспечение безопасности и управление транзакциями.

Общая объектно-ориентированная методика повторного использования общей функциональности заключается в применении наследования или делегирования. Однако наследование может привести к созданию хрупкой иерархии объектов, если по всему приложению используется один и тот же базовый класс, а реализация делегирования может оказаться слишком громоздкой, поскольку может потребоваться выполнять сложные вызовы объекта-делегата.

Аспекты предлагают альтернативу наследованию и делегированию, более простую во многих случаях. При использовании аспектно-ориентированного программирования общая функциональность так же определяется в одном месте, но порядок и место применения этой функциональности можно определять декларативно, без изменения класса, к которому применяются новые возможности. Сквозные функции могут быть выделены в специальные классы, называемые *аспектами*. Это дает два преимущества. Во-первых, логика каждой из них теперь сосредоточена в одном месте, в отличие от случаев, когда она разбросана по всей программе. Во-вторых, упрощаются прикладные модули, поскольку содержат только реализацию своей главной задачи (или базовой функциональности), а вторичные проблемы вынесены в аспекты.

5.1.1. Определение терминологии АОР

Как и большинство технологий, в АОР сформировался свой собственный жаргон. Аспекты часто описываются в терминах «советов», «срезов множества точек сопряжения» и «точек сопряжения». Взаимосвязь этих понятий иллюстрирует рис. 5.2.

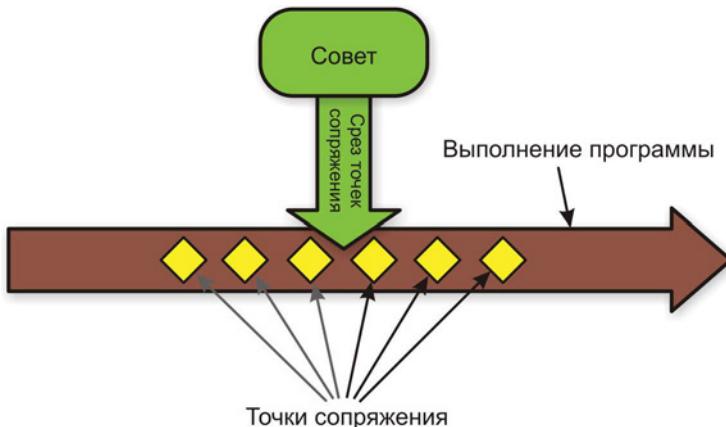


Рис. 5.2. Функциональность аспектов (советов) вплетается в поток выполнения программы в одной или нескольких точках сопряжения

К сожалению, многие термины, используемые для описания АОР, непонятны для непосвященных. Тем не менее они являются частью



идиомы AOP, и знать их совершенно необходимо для понимания AOP. Прежде чем пускаться в путь, нужно научиться говорить.

Совет

Когда учетчик снимает показания электросчетчика, его цель – сообщить в электрическую компанию количество потребленных киловатт. Конечно, он имеет список домов, которые он должен посетить, а информация, сообщаемая им, имеет большое значение. Но основной работой учетчика является снятие показаний электросчетчиков.

Аспекты также имеют свою цель – работу, которую они призваны делать. В терминах AOP работа аспекта называется *совет*.

Совет определяет, *что и когда* делает аспект. В дополнение к описанию работы, выполняемой аспектом, совет учитывает, когда следует ее выполнять. Будет ли она выполняться перед вызовом метода? После его вызова? Или и в том, и другом случаях? Или только когда метод возбудит исключение?

Аспекты Spring могут работать с пятью типами советов:

- до* – работа выполняется перед вызовом метода;
- после* – работа выполняется после вызова метода, независимо от результата;
- после успешного вызова* – работа выполняется после вызова метода, если его выполнение завершилось успешно;
- после исключения* – работа выполняется после того, как вызванный метод возбудит исключение;
- вокруг* – аспект обертывает метод, обеспечивая выполнение некоторых операций до и после вызова метода.

Точки сопряжения

Электрическая компания обслуживает несколько домов, возможно даже целый город. В каждом доме имеется электросчетчик, с которого требуется снимать показания, поэтому каждый дом является потенциальной целью учетчика. Теоретически учетчик способен снимать показания с любых приборов учета, но его работа заключается в снятии показаний только с электросчетчиков.

Также и приложение может иметь тысячи точек применения совета. Эти точки известны как *точки сопряжения* (*join points*). *Точка сопряжения* – это точка в потоке выполнения приложения, куда может быть внедрен аспект. Это может быть вызов метода, возбуждение исключения или даже изменение поля. Все это – точки, куда может быть внедрен аспект для добавления новой особенности поведения.

Срезы множества точек сопряжения

Один учетчик просто физически неспособен посетить все дома, обслуживаемые электрической компанией. Вместо этого компания нанимает штат учетчиков и каждому из них определяет некоторое количество домов, которые он должен посетить. Аналогично от аспекта не требуется воздействовать на все точки сопряжения в приложении. Срезы множества точек сопряжения помогают сузить множество точек для внедрения аспекта.

Если совет отвечает на вопросы *что* и *когда*, то срезы множества точек сопряжения отвечают на вопрос *где*. Срез содержит одну или более точек сопряжения, куда должны быть вплетены советы. Часто срезы множества точек сопряжения определяются за счет явного указания имен классов и методов или через регулярные выражения, определяющие шаблоны имен классов и методов. Некоторые фреймворки, поддерживающие АОР, позволяют создавать срезы множества точек сопряжения динамически, определяя необходимость применения совета, опираясь на решения, принимаемые во время выполнения, такие как значения параметров метода.

Аспекты

В начале рабочего дня учетчик знает, что он должен сделать (составить отчет о потреблении электроэнергии) и какие дома он должен посетить. То есть он знает все, что необходимо знать для выполнения работы.

Аспект объединяет в себе совет и срез множества точек сопряжения. Взятые вместе, они определяют все, что нужно знать об аспекте, – что он делает, где и когда.

Внедрение

Внедрение позволяет добавлять новые методы или атрибуты в существующие классы. Например, можно создать класс-совет `Auditable`, хранящий информацию о том, когда объект был изменен в последний раз. Это может быть очень простой класс, состоящий из единственного метода, например `setLastModified(Date)`, и переменной экземпляра для хранения этой информации. В дальнейшем новый метод и переменная могут быть внедрены в существующие классы без их изменения, добавляя новые черты поведения и информацию.

Вплетение

Вплетение – это процесс применения аспектов к целевому объекту для создания нового, проксируированного объекта. Аспекты вплетаются



в целевой объект в указанные точки сопряжения. Вплетение может происходить в разные моменты жизненного цикла целевого объекта.

- ❑ **Во время компиляции** – аспекты вплетаются в целевой объект, когда тот компилируется. Это требует специального компилятора, такого как AspectJ, вплетающего аспекты на этапе компиляции.
- ❑ **Во время загрузки класса** – вплетение аспектов выполняется в процессе загрузки целевого класса виртуальной машиной JVM. Это требует специального загрузчика, который дополняет байт-код целевого класса перед внедрением его в приложение, например механизм *load-time weaving* (LTW) в AspectJ 5.
- ❑ **Во время выполнения** – вплетение аспектов производится во время выполнения приложения. В этом случае контейнер AOP обычно динамически генерирует объект с вплетенным аспектом, представляющим целевой объект.

Вы познакомились с довольно большим количеством терминов. Если теперь вернуться к рис. 5.2, можно увидеть, как совет, содержащий реализацию сквозной функции, может применяться к объектам в приложении. Точки сопряжения – это все точки в потоке выполнения приложения, к которым при необходимости можно было бы применить совет. Срез множества точек сопряжения определяет, куда (к каким точкам сопряжения) применяется этот совет. Здесь важно понять, что срез определяет точки сопряжения для применения совета.

Теперь, после знакомства с терминологией AOP, посмотрим, как эти основные концепции AOP реализованы в Spring.

5.1.2. Поддержка AOP в Spring

Не все фреймворки AOP равнозначны. Они могут отличаться богатством моделирования точек сопряжения. Некоторые позволяют применять советы на уровне изменения полей, тогда как другие предлагают точки сопряжения на уровне вызовов методов. Они могут также отличаться порядком и особенностями вплетения аспектов. Но не эти особенности делают фреймворки фреймворками с поддержкой AOP, а возможность определения срезов множества точек сопряжения аспектов.

За последние несколько лет в AOP многое изменилось. В результате наведения порядка в этой области некоторые фреймворки объединились, а некоторые исчезли со сцены. В 2005 году произошло слияние проектов AspectWerkz и AspectJ, ознаменовавшее самое

значительное событие в мире AOP и оставившее нам три доминирующих фреймворка AOP:

- AspectJ (<http://eclipse.org/aspectj>);
- JBoss AOP (<http://www.jboss.org/jbossaop>);
- Spring AOP (<http://www.springframework.org>).

Поскольку данная книга посвящена фреймворку Spring, мы сосредоточимся на Spring AOP. Тем не менее проекты Spring и AspectJ тесно взаимодействуют друг с другом, и реализация поддержки AOP в Spring многое заимствует из AspectJ.

Фреймворк Spring поддерживает четыре разновидности AOP:

- классическое аспектно-ориентированное программирование на основе промежуточных объектов;
- аспекты, создаваемые с применением аннотаций @AspectJ;
- аспекты на основе POJO;
- внедрение аспектов AspectJ (доступно во всех версиях Spring).

Первые три разновидности являются вариантами аспектно-ориентированного программирования на основе промежуточных объектов. Соответственно, поддержка AOP в Spring ограничивается перехватом вызовов методов. Если для работы аспекта потребуется нечто более сложное, чем простой перехват вызовов методов (например, перехват вызова конструктора или определение момента изменения значения свойства), следует подумать о возможности реализации аспектов в AspectJ, возможно, через внедрение компонентов Spring в аспекты AspectJ посредством механизма внедрения зависимостей.

Как? В Spring отсутствует поддержка классического AOP? Определение «классический» обычно несет положительную эмоциональную окраску. Классические автомобили, классические турниры по гольфу, классическая кока-кола – все это воспринимается как положительная характеристика.

Но классическая модель аспектно-ориентированного программирования в Spring не отличается положительными качествами. Да, она была хороша в свое время. Но сейчас фреймворк Spring поддерживает более ясные и простые способы работы с аспектами. В сравнении с простым декларативным AOP и AOP на основе аннотаций классическое AOP выглядит громоздким и слишком сложным. Работа непосредственно с объектами ProxyFactoryBean может оказаться слишком утомительной.

Поэтому я решил не включать в это издание обсуждение поддержки классического аспектно-ориентированного программирования в Spring. Если вам действительно интересно знать, как действует данная модель, загляните в два первых издания этой книги. Но я думаю, что новые модели Spring AOP покажутся вам более простыми в использовании.

В этой главе мы исследуем приемы аспектно-ориентированного программирования с использованием фреймворка Spring, но прежде разберемся с некоторыми ключевыми понятиями фреймворка АОР.

Советы в Spring реализуются на языке Java

Все советы, которые вы будете создавать с использованием фреймворка Spring, будут написаны как стандартные Java-классы. То есть при создании аспектов можно пользоваться той же самой интегрированной средой разработки, которая используется для разработки обычного программного кода на Java. К тому же срезы множества точек сопряжения, указывающие, где должны применяться советы, обычно определяются в XML-файле конфигурации Spring.

Сравните это с AspectJ. Сейчас AspectJ поддерживает возможность определения аспектов на основе аннотаций, но это – расширение языка Java. В таком подходе есть свои преимущества и недостатки. Наличие языка АОР предполагает широкие возможности и богатый набор инструментов АОР, однако прежде чем все это использовать, необходимо изучить инструменты и синтаксис.

Советы в Spring – это объекты времени выполнения

При использовании фреймворка Spring, аспекты вплетаются в компоненты во время выполнения посредством обертывания их прокси-классами. Как показано на рис. 5.3, прокси-класс играет роль целевого компонента, перехватывая вызовы методов и передавая эти вызовы целевому компоненту.

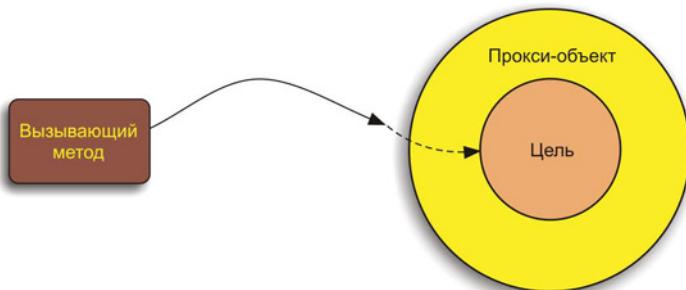


Рис. 5.3. Аспекты Spring реализуются в виде прокси-объектов, обертывающих целевые объекты. Прокси-объект обрабатывает вызовы методов, выполняет дополнительную логику аспекта и затем вызывает целевой метод

Между моментами времени, когда прокси-объект перехватывает вызов метода, и вызовом метода целевого компонента выполняется логика аспекта.

Фреймворк Spring не создает проксированные объекты, пока в них нет необходимости. При использовании контекста приложения ApplicationContext проксированные объекты создаются после загрузки всех компонентов из BeanFactory. Поскольку Spring создает прокси-объекты во время выполнения, для вплетения аспектов в Spring AOP не требуется специальный компилятор.

Spring поддерживает точки сопряжения только для методов

Как упоминалось выше, различные реализации AOP обеспечивают несколько моделей точек сопряжения. Так как поддержка AOP в Spring основана на использовании динамических прокси-объектов, точками сопряжения могут служить только методы, в отличие от некоторых других фреймворков AOP, таких как AspectJ и JBoss, где помимо методов роль точек сопряжения могут играть поля и конструкторы. Отсутствие в Spring возможности применения аспектов к полям препятствует созданию высокоизбирательных аспектов, например для отслеживания изменений полей в объектах. Невозможность применить аспект к конструктору также препятствует реализации логики, которая должна выполняться в момент создания экземпляра компонента.

Однако возможности перехватывать вызовы методов должно быть достаточно для удовлетворения если не всех, то большинства потребностей. Если потребуется нечто большее, чем возможность перехвата вызовов методов, всегда можно воспользоваться расширением AspectJ.

Теперь, получив общее представление о возможностях AOP и особенностях его поддержки в Spring, можно перейти к практическому изучению принципов создания аспектов в Spring. Сначала познакомимся поближе с декларативной моделью AOP.

5.2. Выбор точек сопряжения в описаниях срезов

Как упоминалось выше, срезы множества точек сопряжения – это множества точек применения советов аспектов в программном коде. Вместе с советами аспектов срезы являются одними из наиболее фундаментальных элементов аспектов. Поэтому важно знать и понимать, как они определяются.



В Spring AOP срезы множества точек сопряжения определяются на языке выражений AspectJ. Знакомые с расширением AspectJ не будут испытывать затруднений при определении срезов в Spring. Но если вы незнакомы с AspectJ, этот раздел послужит вам кратким учебником по определению срезов в стиле AspectJ. За более подробной информацией о расширении AspectJ и языке выражений AspectJ я настоятельно рекомендую обратиться к книге Рамниваса Ладдада (Ramnivas Laddad) «AspectJ in Action, Second Edition».

Самое важное, что следует помнить о механизме формирования срезов множества точек сопряжения в стиле AspectJ, входящем в состав Spring AOP, – он поддерживает ограниченное подмножество конструкций описания, доступных в AspectJ. Напомню, что модель Spring AOP основана на использовании прокси-объектов, а некоторые элементы выражений неприменимы в этой модели АОР. В табл. 5.1 перечислены указатели для использования в описаниях срезов в AspectJ, поддерживаемые в Spring AOP.

Таблица 5.1. Конструкции языка выражений описания срезов множества точек сопряжения в AspectJ, поддерживаемые аспектами в Spring

Указатель AspectJ	Описание
args()	Ограничивает срез точек сопряжения вызовами методов, чьи аргументы являются экземплярами указанных типов
@args()	Ограничивает срез точек сопряжения вызовами методов, чьи аргументы аннотированы указанными типами аннотаций
execution()	Соответствует точкам сопряжения, которые являются вызовами методов
this()	Ограничивает срез точек сопряжений точками, где ссылка на компонент является ссылкой на прокси-объект указанного типа
target()	Ограничивает срез точек сопряжений точками, где целевой объект имеет указанный тип
@target()	Ограничивает срез точек сопряжений точками, где класс выполняемого объекта снабжен аннотацией указанного типа
within()	Ограничивает срез точек сопряжений точками только внутри указанных типов
@within()	Ограничивает срез точек сопряжений точками внутри указанных типов, снабженных указанной аннотацией (в Spring AOP соответствует вызовам методов в указанном типе, отмеченных указанной аннотацией)
@annotation	Ограничивает срез точек сопряжений точками, помеченными указанной аннотацией

Попытки использовать любые другие указатели AspectJ приведут к исключению `IllegalArgumentException`.

Обратите внимание, что среди поддерживаемых указателей только `execution` фактически выполняет сопоставление – все остальные используются для ограничения множества совпадений. Это означает, что `execution` является основным указателем, который должен использоваться во всех определениях срезов множества точек сопряжения. Остальные указатели применяются только для ограничения точек сопряжения в срезе.

5.2.1. Определение срезов множества точек сопряжения

На рис. 5.4 представлено выражение, определяющее срез множества точек сопряжения, который можно использовать для применения совета к вызову метода `play()` интерфейса `Instrument`.



Рис. 5.4. Выражение, определяющее срез множества точек сопряжения AspectJ и выбирающее метод `play()` интерфейса `Instrument`

Указатель `execution()` используется здесь для выбора метода `play()` интерфейса `Instrument`. Описание метода начинается со звездочки, указывающей, что тип возвращаемого методом значения не должен учитываться. Далее определяются полное имя класса и имя метода, который требуется выбрать. В списке параметров метода указаны две точки, идущие подряд (...), указывающие, что в срез может быть включена любая точка сопряжения, соответствующая любому вызову метода `play()`, независимо от конкретного списка передаваемых ему аргументов.

Теперь предположим, что необходимо ограничить множество точек сопряжения границами пакета `com.springinaction.springidol`. В этом случае можно задействовать указатель `within()`, как показано на рис. 5.5.



Рис. 5.5. Ограничение множества точек сопряжения с помощью указателя `within()`

Обратите внимание на оператор `&&`, объединяющий указатели `execution()` и `within()` логической операцией «И» (то есть использовать будут только точки сопряжения, где будут выполнены требования обоих указателей). Аналогично, чтобы объединить указатели логической операцией «ИЛИ», можно было бы использовать оператор `||`. А чтобы инвертировать смысл указателя – использовать оператор `!`.

Поскольку в языке разметки XML амперсанды имеют специальное значение, при определении срезов множества точек сопряжения в конфигурационном XML-файле вместо оператора `&&` можно использовать оператор `and`. Аналогично можно использовать операторы `or` и `not` вместо `||` и `!` (соответственно).

5.2.2. Использование указателя `bean()`

Помимо указателей, перечисленных в табл. 5.1, в версии Spring 2.5 появился новый указатель `bean()`, позволяющий идентифицировать компоненты внутри выражений определения срезов по их идентификаторам. Указатель `bean()` принимает идентификатор или имя компонента в виде аргумента и ограничивает срез множества точек сопряжения, оставляя в нем только точки, соответствующие указанному компоненту.

Например, взгляните на следующее определение среза:

```
execution(* com.springinaction.springidol.Instrument.play())
  and bean(eddie)
```

Это определение говорит, что совет аспекта должен применяться к вызовам метода `Instrument.play()`, но только внутри компонента с идентификатором `eddie`.

Возможность ограничения доступных точек сопряжения границами определенного компонента может оказаться ценной в некоторых

случаях, однако имеется также возможность инвертировать условие и обеспечить применение аспекта ко всем компонентам, кроме имеющего определенный идентификатор:

```
execution(* com.springinaction.springidol.Instrument.play())
        and !bean(eddie)
```

В данном случае совет аспекта будет вплетен во все компоненты, кроме компонента с идентификатором `eddie`.

Теперь, после знакомства с основами определения срезов множества точек сопряжения, можно перейти к созданию советов и объявлению аспектов, использующих эти срезы.

5.3. Объявление аспектов в XML

Знакомые с классической моделью аспектно-ориентированного программирования в Spring знают, что работать с `ProxyFactoryBean` очень неудобно. В свое время разработчики Spring осознали это и приступили к реализации более удобного способа объявления аспектов в Spring. В результате их усилий в пространстве имен `aop` появились новые элементы. Перечень элементов настройки механизма AOP приводится в табл. 5.2.

Таблица 5.2. Элементы настройки механизма AOP в Spring упрощают объявление аспектов, основанных на POJO

Элемент настройки AOP	Назначение
<code><aop:advisor></code>	Определяет объект-советник
<code><aop:after></code>	Определяет AOP-совет, выполняемый после вызова метода (независимо от успешности его завершения)
<code><aop:after-returning></code>	Определяет AOP-совет, выполняемый после успешного выполнения метода
<code><aop:after-throwing></code>	Определяет AOP-совет, выполняемый после возбуждения исключения
<code><aop:around></code>	Определяет AOP-совет, выполняемый до и после выполнения метода
<code><aop:aspect></code>	Определяет аспект
<code><aop:aspectj-autoproxy></code>	Включает поддержку аспектов, управляемых аннотациями, созданными с применением аннотации <code>@AspectJ</code>
<code><aop:before></code>	Определяет AOP-совет, выполняемый до выполнения метода

**Таблица 5.2 (окончание)**

Элемент настройки AOP	Назначение
<aop:config>	Элемент верхнего уровня настройки механизма AOP
<aop:declare-parents>	Внедряет в объекты прозрачную реализацию дополнительных интерфейсов
<aop:pointcut>	Определяет срез точек сопряжения

В главе 2 демонстрировалось внедрение зависимостей на примере реализации конкурса талантов «Spring Idol». В этом примере было подключено нескольких исполнителей в виде компонентов Spring, чтобы они могли продемонстрировать свои таланты. Это довольно забавный пример. Но для конкурсов, подобных этому, нужны зрители, иначе в их проведении нет никакого смысла.

Поэтому, чтобы продемонстрировать возможности Spring AOP, создадим класс Audience для нашего примера конкурса талантов. В листинге 5.1 представлен класс, определяющий функции зрителей.

Листинг 5.1. Класс Audience для конкурса талантов

```
package com.springinaction.springidol;

public class Audience {
    public void takeSeats() { // Перед выступлением
        System.out.println("The audience is taking their seats.");
    }

    public void turnOffCellPhones() { // Перед выступлением
        System.out.println("The audience is turning off their cellphones");
    }

    public void applaud() { // После выступления
        System.out.println("CLAP CLAP CLAP CLAP CLAP");
    }

    public void demandRefund() { // После неудачного выступления
        System.out.println("Boo! We want our money back!");
    }
}
```

Как видите, в классе Audience нет ничего примечательного. Это обычный Java-класс с горсткой методов. Его можно зарегистриро-

вать в виде компонента в контексте приложения Spring, подобно любому другому классу:

```
<bean id="audience"
      class="com.springinaction.springidol.Audience" />
```

Несмотря на его неприметность, класс Audience обладает всем необходимым для создания аспекта. К нему нужно лишь добавить немного волшебства Spring AOP.

5.3.1. Объявление советов, выполняемых до или после

С помощью элементов настройки механизма Spring AOP компонент audience можно превратить в аспект, как показано в листинге 5.2.

Листинг 5.2. Определение аспекта audience с использованием элементов настройки Spring AOP

```
<aop:config>
    <aop:aspect ref="audience">           <!-- Ссылка на компонент audience -->

        <aop:before pointcut=
            "execution(* com.springinaction.springidol.Performer.perform(..))"
            method="takeSeats" />           <!-- Перед выступлением -->

        <aop:before pointcut=
            "execution(* com.springinaction.springidol.Performer.perform(..))"
            method="turnOffCellPhones" />   <!-- Перед выступлением -->

        <aop:after-returning pointcut=
            "execution(* com.springinaction.springidol.Performer.perform(..))"
            method="applaud" />           <!-- После выступления -->

        <aop:after-throwing pointcut=
            "execution(* com.springinaction.springidol.Performer.perform(..))"
            method="demandRefund" />       <!-- После неудачного выступления -->

    </aop:aspect>
</aop:config>
```

Первое, на что следует обратить внимание, – большинство элементов настройки должны находиться внутри элемента `<aop:config>`.

Есть некоторые исключения из этого правила, но когда дело доходит до объявления компонентов аспектами, конфигурация всегда должна начинаться с элемента `<aop:config>`.

Внутри `<aop:config>` можно объявить один или более объектов-советников, аспектов или срезов множества точек сопряжения. В листинге 5.2 был объявлен единственный аспект с помощью элемента `<aop:aspect>`. Атрибут `ref` ссылается на компонент POJO, реализующий функциональность аспекта, в данном случае `Audience`. Компонент, на который ссылается атрибут `ref`, определяет методы, вызываемые советами в аспекте.

Сам аспект определяет четыре совета. Два элемента `<aop:before>` определяют советы, выполняемые *перед вызовом метода*, которые будут вызывать методы `takeSeats()` и `turnOffCellPhones()` компонента `Audience` перед вызовами любых методов, определяемых срезом с точками сопряжения. Элемент `<aop:after-returning>` определяет совет, выполняемый *после успешного завершения метода* и вызывающий метод `applaud()` после любых методов, определяемых срезом. И элемент `<aop:after-throwing>` определяет совет, выполняемый *в случае возбуждения исключения* и вызывающий метод `demandRefund()`, если было возбуждено какое-либо исключение. На рис. 5.6 иллюстрируется, как логика советов вплетается в основную логику работы приложения.

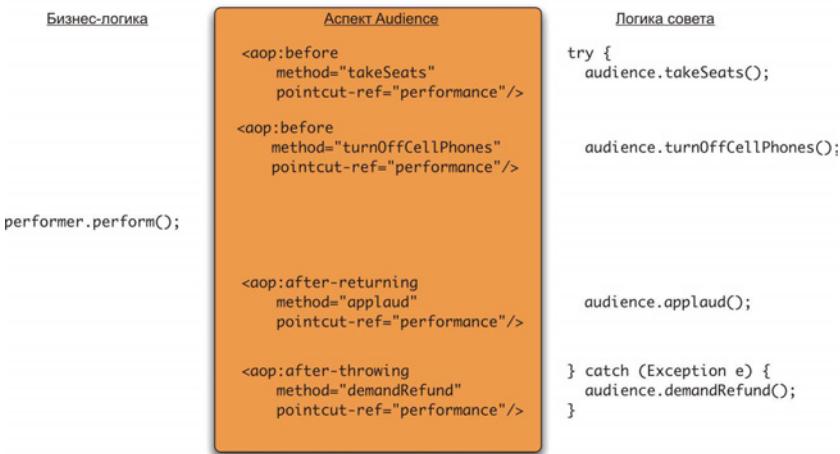


Рис. 5.6. Аспект `Audience` включает четыре совета, вплетаемых в логику вызова методов, определяемых срезом с точками сопряжения

Во всех элементах, объявляющих советы, атрибут `pointcut` – срез множества точек сопряжения, где будет применяться данный совет. Значением атрибута `pointcut` является определение среза с применением синтаксиса выражений AspectJ.

Обратите внимание, что во всех элементах объявления советов атрибут `pointcut` имеет одинаковое значение, потому что все советы применяются к одному и тому же срезу множества точек сопряжения.

Однако это является нарушением принципа DRY(don't repeat yourself – не повторяйся). Если впоследствии потребуется изменить определение среза, это придется сделать в четырех различных местах.

Чтобы избежать дублирования определений срезов множества точек сопряжения, можно определить именованный срез с помощью элемента `<aop:pointcut>`. В листинге 5.3 представлен фрагмент XML, демонстрирующий, как внутри элемента `<aop:aspect>` определить именованный срез с помощью элемента `<aop:pointcut>`, который затем можно использовать во всех элементах определения советов.

Листинг 5.3. Определение именованного среза множества точек сопряжения для устранения избыточных определений срезов

```
<aop:config>
    <aop:aspect ref="audience">
        <aop:pointcut id="performance" expression=
            "execution(* com.springinaction.springidol.Performer.perform(..))"
        /> <!-- Определение среза множества точек сопряжения -->

        <aop:before
            pointcut-ref="performance"      <!-- Ссылка на именованный срез -->
            method="takeSeats" />

        <aop:before
            pointcut-ref="performance"      <!-- Ссылка на именованный срез -->
            method="turnOffCellPhones" />

        <aop:after-returning
            pointcut-ref="performance"      <!-- Ссылка на именованный срез -->
            method="applaud" />

        <aop:after-throwing
            pointcut-ref="performance"      <!-- Ссылка на именованный срез -->
            method="demandRefund" />

    </aop:aspect>
</aop:config>
```



Теперь определение среза, ссылки на который используются в нескольких местах, существует в единственном экземпляре. Элемент `<aop:pointcut>` определяет срез множества точек сопряжения с идентификатором `performance`. Кроме того, во всех элементах, объявляющих советы, определения срезов были заменены атрибутом `pointcut-ref` со ссылкой на именованный срез.

Как показано в листинге 5.3, элемент `<aop:pointcut>` определяет срез множества точек сопряжения, на который можно ссылаться из любых советов, объявленных внутри того же элемента `<aop:aspect>`. Однако имеется возможность определять срезы, на которые можно ссылаться из разных аспектов, для чего достаточно поместить элементы `<aop:pointcut>` в область действия элемента `<aop:config>`.

5.3.2. Объявление советов, выполняемых и до, и после

Текущая реализация класса `Audience` работает замечательно. Но простые советы, выполняемые до или после вызова метода, имеют некоторые ограничения. В частности, довольно сложно обеспечить совместное использование информации советами, выполняемыми до или после, не прибегая к использованию переменных экземпляра.

Например, предположим, что к выключению сотовых телефонов и аплодисментов в конце также было бы желательно, чтобы зрители посматривали на часы и отмечали продолжительность выступления. Единственный способ решить эту задачу – использовать совет, выполняемый до и после, чтобы перед выступлением засечь текущее время, а после выступления определить его продолжительность. Но при этом придется сохранить время начала в переменной экземпляра. Поскольку экземпляр класса `Audience` – единственный в приложении, он не может использоваться в многопоточной среде выполнения без риска повредить хранимые в нем данные.

В этом отношении совет, выполняемый и до, и после вызова целевого метода, имеет преимущество перед советами, выполняемыми до или после. Он позволяет решать те же задачи, что и советы, выполняемые до или после, но делает это в единственном методе. А так как совет целиком выполняется в рамках единственного метода, нет необходимости сохранять промежуточные данные в переменных экземпляра.

Например, взгляните на новый метод `watchPerformance()`, представленный в листинге 5.4.

Листинг 4.4. Метод watchPerformance() реализует совет AOP, выполняемый и до, и после вызова целевого метода

```
public void watchPerformance(ProceedingJoinPoint joinpoint) {  
    try {  
        System.out.println("The audience is taking their seats.");  
        System.out.println("The audience is turning off their cellphones");  
        long start = System.currentTimeMillis(); // Перед выступлением  
  
        joinpoint.proceed(); // Вызов целевого метода  
  
        long end = System.currentTimeMillis(); // После выступления  
        System.out.println("CLAP CLAP CLAP CLAP");  
        System.out.println("The performance took " + (end - start)  
            + " milliseconds.");  
    } catch (Throwable t) {  
        System.out.println("Boo! We want our money back!");  
    }  
}
```

Первое, на что следует обратить внимание в реализации нового совета, – он принимает параметр типа `ProceedingJoinPoint`. Этот объект необходим для вызова целевого метода внутри совета. Метод совета выполнит все необходимые предварительные операции и, когда будет готов передать управление целевому методу, вызовет метод `ProceedingJoinPoint.proceed()`.

Имейте в виду, что крайне важно не забывать включать вызов метода `proceed()`. Если этого не сделать, совет фактически заблокирует доступ к целевому методу. Возможно, в некоторых случаях именно это и требуется, но чаще бывает желательно, чтобы целевой метод был выполнен в какой-то момент.

Интересно отметить, что, кроме блокирования доступа к целевому методу отказом от вызова метода `proceed()`, внутри совета его также можно вызвать несколько раз. Это может пригодиться для реализации повторных попыток вызова целевого метода, на случай если он может потерпеть неудачу.

В примере с аспектом `audience` метод `watchPerformance()` реализует все, что было реализовано в четырех прежних методах совета, но теперь все это сосредоточено в единственном методе, и он несет полную ответственность за обработку ошибок. Обратите также внимание, что непосредственно перед вызовом метода `proceed()`, соответствующего точке сопряжения, в локальной переменной сохра-



няется текущее время. Сразу после возврата из метода выводится длительность выступления.

Объявление совета, выполняющегося и до, и после целевого метода, мало отличается от объявлений советов других типов, достаточно просто использовать элемент `<aop:around>`, как показано в листинге 5.5.

Листинг 5.5. Определение аспекта audience с единственным советом, выполняемым и до, и после вызова целевого метода

```
<aop:config>
    <aop:aspect ref="audience">
        <aop:pointcut id="performance2" expression=
            "execution(* com.springinaction.springidol.Performer.perform(..))"
        />
        <!-- Совет, выполняемый и до, и после -->
        <aop:around
            pointcut-ref="performance2"
            method="watchPerformance()" />
    </aop:aspect>
</aop:config>
```

Подобно другим XML-элементам определения советов, в элементе `<aop:around>` указываются срез множества точек сопряжения и имя метода, реализующего совет. Здесь используется тот же срез, что и прежде, но в атрибуте `method` указано имя нового метода `watchPerformance()`.

5.3.3. Передача параметров советам

До сих пор наши аспекты отличались простотой реализации и не принимали параметров. Единственное исключение – метод `watchPerformance()`, реализующий совет, выполняемый и до, и после целевого метода, который принимает параметр типа `ProceedingJoinPoint`. Реализованный нами совет никак не заботится о параметрах для передачи целевому методу. Впрочем, в этом нет ничего страшного, потому что вызываемый нами метод `perform()` не принимает никаких параметров.

Тем не менее иногда бывает желательно, чтобы совет не только обертывал целевой метод, но также проверял значения передаваемых ему параметров.

Чтобы увидеть, как действует эта возможность, познакомимся с новым участником конкурса «Spring Idol». Наш новый участник обладает талантом чтения мыслей и определяется интерфейсом `MindReader`:

```
package com.springinaction.springidol;

public interface MindReader {
    void interceptThoughts(String thoughts);

    String getThoughts();
}
```

Человек, читающий мысли, выполняет два основных задания: он читает мысли добровольца и сообщает их публике. Ниже представлен простой класс Magician, реализующий интерфейс MindReader:

```
package com.springinaction.springidol;

public class Magician implements MindReader {
    private String thoughts;

    public void interceptThoughts(String thoughts) {
        System.out.println("Intercepting volunteer's thoughts");
        this.thoughts = thoughts;
    }

    public String getThoughts() {
        return thoughts;
    }
}
```

Теперь нам нужен доброволец, чьи мысли будут прочитаны. Для этой цели определим интерфейс Thinker:

```
package com.springinaction.springidol;

public interface Thinker {
    void thinkOfSomething(String thoughts);
}
```

Класс Volunteer представляет собой простейшую реализацию интерфейса Thinker:

```
package com.springinaction.springidol;

public class Volunteer implements Thinker {
    private String thoughts;
```

```
public void thinkOfSomething(String thoughts) {  
    this.thoughts = thoughts;  
}  
  
public String getThoughts() {  
    return thoughts;  
}  
}
```

Детали реализации Volunteer не представляют особого интереса или важности. Что действительно интересно – как фокусник (объект Magician) будет читать мысли добровольца (объект Volunteer) с использованием Spring AOP.

Для осуществления телепатического контакта воспользуемся теми же элементами <aop:aspect> и <aop:before>, что и прежде. Но на этот раз настроим в них передачу совету параметров для целевого метода.

```
<aop:config>  
    <aop:aspect ref="magician">  
        <aop:pointcut id="thinking"  
            expression="execution(*  
com.springinaction.springidol.Thinker.thinkOfSomething(String))  
                and args(thoughts)" />  
  
        <aop:before  
            pointcut-ref="thinking"  
            method="interceptThoughts"  
            arg-names="thoughts" />  
    </aop:aspect>  
</aop:config>
```

Ключ к телепатическим возможностям фокусника Magician находится в определении среза множества точек сопряжения и в атрибуте arg-names элемента <aop:before>. Срез идентифицирует метод thinkOfSomething() интерфейса Thinker, указывая, что метод принимает строковый аргумент, и уточняя имя аргумента thoughts с помощью указателя args.

Элемент <aop:before> определения совета также ссылается на аргумент thoughts, указывая, что он должен передаваться методу interceptThoughts() класса Magician.

Теперь всякий раз, когда будет вызываться метод thinkOfSomething() компонента volunteer, аспект Magician будет перехватывать его мысли.

Чтобы убедиться в этом, ниже представлен простой тестовый класс со следующим методом:

```
@Test
public void magicianShouldReadVolunteersMind() {
    volunteer.thinkOfSomething("Queen of Hearts");

    assertEquals("Queen of Hearts", magician.getThoughts());
}
```

Подробнее о создании модульных и интеграционных тестов в Spring будет рассказываться в следующей главе. А пока просто отметьте, что тест выполняется без ошибок, потому что фокусник всегда будет знать, о чём думает доброволец.

Теперь посмотрим, как с помощью Spring AOP добавлять новые функциональные возможности в существующие объекты.

5.3.4. Внедрение новых возможностей с помощью аспектов

В некоторых языках программирования, таких как Ruby и Groovy, есть понятие открытых классов. Они позволяют добавлять в объекты или классы новые методы без непосредственного изменения определения этих объектов/классов. К сожалению, язык Java не обладает такими динамическими возможностями. Как только класс будет скомпилирован, у вас остается совсем немного возможностей расширить его функциональные возможности.

Но если задуматься, разве аспекты, рассматривавшиеся в этой главе, не являются такой возможностью? Конечно, мы не добавляли новых методов в объекты, но мы расширяли функциональные возможности уже существующих методов. Коль скоро аспекты позволяют обертывать существующие методы новыми функциональными возможностями, почему бы с их помощью не попробовать добавлять новые методы? В действительности концепция *внедрения*, существующая в аспектно-ориентированном программировании, позволяет присоединять новые методы к компонентам Spring.

Напомню, что аспекты в Spring – это всего лишь промежуточные объекты-обертки (прокси-объекты), реализующие те же самые интерфейсы, что и компоненты, которые они обертывают. Что, если кроме реализации этих интерфейсов прокси-объект будет содержать реализацию некоторого другого интерфейса? Тогда любой



компонент, окруженный таким аспектом, будет выглядеть как объект, реализующий дополнительный интерфейс, даже если лежащий в его основе класс в действительности не реализует его. На рис. 5.7 изображено, как действует этот механизм.

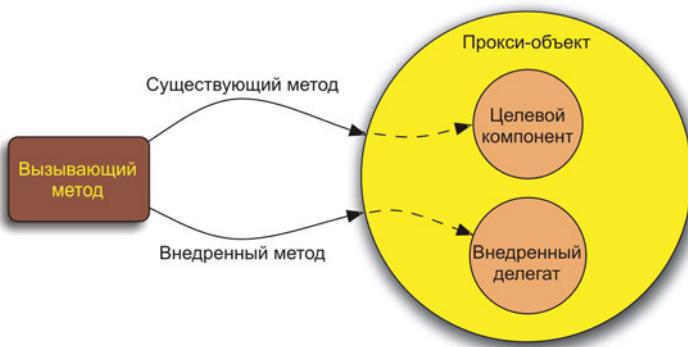


Рис. 5.7. С помощью Spring AOP
можно внедрять новые методы в компоненты.
Прокси-объект перехватывает вызовы и делегирует их
другому объекту, реализующему новый метод

На рис. 5.7 видно, что при вызове метода внедренного интерфейса прокси-объект делегирует вызов некоторому другому объекту, содержащему реализацию нового интерфейса. Фактически это позволяет разбить реализацию компонента на несколько классов.

Попробуем воплотить эту идею на практике. Представим, что нам потребовалось внедрить следующий интерфейс `Contestant` во все компоненты, представляющие участников конкурса в примере:

```
package com.springinaction.springidol;

public interface Contestant {
    void receiveAward();
}
```

Я понимаю, что можно было бы обойти все реализации интерфейса `Performer` и изменить их, добавив реализацию интерфейса `Contestant` (соперник). Но в общем и целом это может быть не самый благородный шаг (потому что `Contestant` (соперник) и `Performer` (исполнитель) – необязательно взаимоисключающие понятия). Бо-

лее того, иногда может оказаться просто невозможным изменить все реализации интерфейса `Performer`, особенно когда в приложении используются сторонние реализации и отсутствуют их исходные тексты.

К счастью, возможность внедрения, поддерживаемая АОР, в состоянии помочь решить эту проблему без изменения архитектурных решений и не требуя изменять существующие реализации. Для осуществления описанной задачи необходимо задействовать элемент `<aop:declare-parents>`:

```
<aop:aspect>
    <aop:declare-parents
        types-matching="com.springinaction.springidol.Performer+"
        implement-interface="com.springinaction.springidol.Contestant"
        default-impl="com.springinaction.springidol.GraciousContestant"
    />
</aop:aspect>
```

Как следует из его имени, элемент `<aop:declare-parents>` объявляет, что компоненты, к которым применяется описываемый аспект, приобретают новых родителей в иерархии наследования. В частности, в данном случае утверждается, что компоненты, чьи типы совместимы с интерфейсом `Performer` (определяется атрибутом `types-matching`), получают в иерархии интерфейс `Contestant` (определяется атрибутом `implement-interface`) в иерархии наследования. Последний атрибут элемента описывает, где находятся реализации методов интерфейса `Contestant`.

Существуют два способа определения реализации внедренного интерфейса. В данном случае был использован атрибут `default-impl`, явно определяющий реализацию посредством полного имени класса. Другой способ заключается в использовании атрибута `delegate-ref`:

```
<aop:declare-parents
    types-matching="com.springinaction.springidol.Performer+"
    implement-interface="com.springinaction.springidol.Contestant"
    delegate-ref="contestantDelegate"
/>
```

Атрибут `delegate-ref` ссылается на компонент Spring, играющий роль внедренного делегата. Он предполагает существование компонента с идентификатором `contestantDelegate` в контексте Spring:



```
<bean id="contestantDelegate"
      class="com.springinaction.springidol.GraciousContestant" />
```

Разница между двумя способами определения делегата с помощью атрибутов `default-impl` и `delegate-ref` заключается в том, что во втором случае компонент Spring сам может быть субъектом внедрения, применения аспектов и воздействия других механизмов Spring.

5.4. Аннотирование аспектов

Важной особенностью, появившейся в AspectJ 5, стала возможность использовать аннотации для создания аспектов. До выхода версии AspectJ 5 для создания аспектов AspectJ требовалось знать синтаксис расширения языка Java. Однако появление возможности использования аннотаций AspectJ свело преобразование любых классов в аспекты к простому добавлению в них нескольких аннотаций. Эту новую возможность часто называют как `@AspectJ`.

Взглянув еще раз на класс `Audience`, можно заметить, что он содержит всю необходимую функциональность, реализующую поведение публики в зале, но в нем отсутствует что-либо, превращающее его в аспект. По этой причине нам пришлось объявлять совет и срез множества точек сопряжения в конфигурационном XML-файле.

С помощью аннотаций `@AspectJ` можно превратить класс `Audience` в аспект, не прибегая к созданию дополнительных классов или объявлению компонентов. В листинге 5.6 представлен новый класс `Audience`, на этот раз преобразованный в аспект с помощью аннотаций.

Листинг 5.6. Преобразование класса Audience в аспект с помощью аннотаций

```
package com.springinaction.springidol;

import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Audience {
    @Pointcut( // Определение среза
```

```
"execution(* com.springinaction.springidol.Performer.perform(..))")
public void performance() {
}

@Before("performance()")
public void takeSeats() {           // Перед выступлением
    System.out.println("The audience is taking their seats.");
}

@Before("performance()")           // Перед выступлением
public void turnOffCellPhones() {
    System.out.println("The audience is turning off their cellphones");
}

@AfterReturning("performance()")    // После успешного выступления
public void applaud() {
    System.out.println("CLAP CLAP CLAP CLAP CLAP");
}

@AfterThrowing("performance()")
public void demandRefund() {        // После неудачного выступления
    System.out.println("Boo! We want our money back!");
}
```

Теперь новый класс Audience отмечен аннотацией @Aspect. Эта аннотация указывает, что Audience является не обычным простым Java-объектом, а аспектом.

Аннотация @Pointcut используется в @AspectJ для определения среза множества точек сопряжения. Значение параметра аннотации @Pointcut является выражением AspectJ. Здесь выражение указывает, что аспект будет подключаться к точкам сопряжения, соответствующим методу perform() интерфейса Performer. Имя среза определяется именем метода, к которому применяется аннотация. То есть данный срез имеет имя performance(). Фактическое тело метода performance() не имеет значения и в действительности должно быть пустым. Сам метод служит лишь точкой подключения аннотации @Pointcut.

Все методы класса Audience отмечены аннотациями, определяющими советы. Аннотация @Before применяется к методам takeSeats() и turnOffCellPhones(), указывая, что эти два метода являются советами, выполняемыми перед вызовом целевого метода. Аннотация @AfterReturning указывает, что метод applaud() является советом, вы-



полняемым в случае благополучного завершения целевого метода. И аннотация `@AfterThrowing` перед методом `demandRefund()` указывает, что он будет вызываться в случае появления исключительных ситуаций в процессе выступления.

Всем аннотациям советов передается строка с именем среза множества точек сопряжения `performance()`, определяющая точки применения советов.

Обратите внимание, что кроме добавления аннотаций и пустого метода `performance()` реализация класса `Audience` осталась прежней. Это означает, что он все еще остается простым Java-объектом и может использоваться в этом его качестве. Он также может быть связан в конфигурации Spring, как показано ниже:

```
<bean id="audience"
      class="com.springinaction.springidol.Audience" />
```

Поскольку класс `Audience` содержит собственные определения множества точек сопряжения и советов, больше нет необходимости объявлять их в конфигурационном XML-файле. Последнее, что осталось сделать, – это применить `Audience` как аспект. Для этого необходимо объявить в контексте Spring компонент, реализующий автоматическое проксирование, который знает, как превратить компоненты, отмеченные аннотациями `@AspectJ`, в советы.

Для этой цели в состав Spring входит класс `AnnotationAwareAspectJAutoProxyCreator`, создающий объекты автоматического проксирования. Класс `AnnotationAwareAspectJAutoProxyCreator` можно зарегистрировать в контексте Spring с помощью элемента `<bean>`, но это потребует большого объема ввода с клавиатуры (уж поверьте... мне приходилось делать это несколько раз). Чтобы упростить эту задачу, фреймворк Spring предоставляет собственный конфигурационный элемент в пространстве имён `aop`, гораздо более простой в запоминании, нежели это длинное имя класса:

```
<aop:aspectj-autoproxy />
```

Элемент `<aop:aspectj-autoproxy>` создаст в контексте Spring компонент класса `AnnotationAwareAspectJAutoProxyCreator` и автоматически выполнит проксирование компонентов, имеющих методы, совпадающие с объявлениями точек сопряжения в аннотациях `@Pointcut`, которые присутствуют в компонентах, отмеченных аннотацией `@Aspect`.

Перед использованием элемента `<aop:aspectj-autoproxy>` нужно не забыть включить пространство имен `aop` в конфигурационном файле Spring:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

Вы должны понимать, что в качестве руководства к действию при создании аспектов элемент `<aop:aspectj-autoproxy>` использует только аннотации `@AspectJ`. В действительности эти аспекты все еще остаются аспектами Spring, а это означает, что, несмотря на использование аннотаций `@AspectJ`, советы могут применяться только к вызовам методов. Если необходимы дополнительные возможности `AspectJ`, следует использовать механизм `AspectJ` времени выполнения и не полагаться на возможность фреймворка Spring создавать аспекты на основе прокси-объектов.

Важно также отметить, что и элемент `<aop:aspect>`, и аннотации `@AspectJ` обеспечивают эффективные способы превращения РОJO в аспекты. Но элемент `<aop:aspect>` обладает одним важным преимуществом перед аннотациями `@AspectJ` – он не требует изменения исходного программного кода класса для преобразования его в аспект. Для применения аннотаций `@AspectJ` к классам и методам необходимо иметь исходные тексты, тогда как элемент `<aop:aspect>` может ссылаться на любой компонент.

Теперь посмотрим, как с помощью аннотаций `@AspectJ` создать совет, выполняемый до, и до, и после вызова целевого метода.

5.4.1. Создание советов, выполняемых и до, и после

Как и в случае настройки аспектов в XML-файле конфигурации Spring, при использовании аннотаций `@AspectJ` можно создавать не только советы, выполняемые до или после вызова целевого метода, но и советы, выполняемые и до, и после вызова. Для этого следует использовать аннотацию `@Around`, как показано в следующем примере:

```
@Around("performance()")
public void watchPerformance(ProceedingJoinPoint joinpoint) {
    try {
        System.out.println("The audience is taking their seats.");
        System.out.println("The audience is turning off their cellphones");

        long start = System.currentTimeMillis();
        joinpoint.proceed();
        long end = System.currentTimeMillis();

        System.out.println("CLAP CLAP CLAP CLAP CLAP");

        System.out.println("The performance took " + (end - start)
            + " milliseconds.");
    } catch (Throwable t) {
        System.out.println("Boo! We want our money back!");
    }
}
```

Здесь аннотация @Around указывает, что метод watchPerformance() используется как совет, выполняемый и до, и после вызова целевого метода, соответствующего срезу множества точек сопряжения performance(). Этот фрагмент может показаться до боли знакомым, что неудивительно, так как это тот же самый метод watchPerformance(), который мы видели выше. Только на этот раз он отмечен аннотацией @Around.

Как вы помните, методы советов, выполняемых и до, и после вызова целевого метода, должны явно вызывать метод proceed(), чтобы выполнить целевой метод. Но простого применения аннотации @Around к методу недостаточно, чтобы обеспечить вызов метода proceed(). Поэтому методы, реализующие советы, выполняемые и до, и после вызова целевого метода, должны принимать в виде аргумента объект типа ProceedingJoinPoint и вызывать метод proceed() этого объекта.

5.4.2. Передача аргументов аннотированным советам

Передача параметров советам, созданным с помощью аннотаций @AspectJ, мало чем отличается от случая, когда аспекты объявляются в конфигурационном XML-файле Spring. В действительности

XML-элементы, использовавшиеся выше, имеют прямые эквиваленты в виде аннотаций @AspectJ, как показано ниже, на примере нового класса Magician.

Листинг 5.7. Превращение класса Magician в аспект с помощью аннотаций @AspectJ

```
package com.springinaction.springidol;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class Magician implements MindReader {
    private String thoughts;

    // Объявление параметризованного среза множества точек сопряжения
    @Pointcut("execution(* com.springinaction.springidol."
        + "Thinker.thinkOfSomething(String)) && args(thoughts)")
    public void thinking(String thoughts) {
    }

    @Before("thinking(thoughts)")           // Передача параметра в совет
    public void interceptThoughts(String thoughts) {
        System.out.println("Intercepting volunteer's thoughts : "
            + thoughts);
        this.thoughts = thoughts;
    }

    public String getThoughts() {
        return thoughts;
    }
}
```

Элемент <aop:pointcut> является эквивалентом аннотации @Pointcut, а элемент <aop:before> – эквивалентом аннотации @Before. Единственное важное отличие состоит в том, что аннотации @AspectJ могут опираться на синтаксис языка Java для объявления значений параметров, передаваемых советам. Поэтому здесь нет никакой потребности в наличии прямого эквивалента атрибута arg-names элемента <aop:before>.



5.4.3. Внедрение с помощью аннотаций

Выше было показано, как с помощью элемента `<aop:declare-parents>` внедрить реализацию интерфейса в существующий компонент, не изменяя исходного программного кода. Теперь посмотрим на этот же пример с другой стороны, но на этот раз задействуем аннотации AOP.

Эквивалентом элемента `<aop:declare-parents>` в множестве аннотаций `@AspectJ` является аннотация `@DeclareParents`. Она действует практически так же, как и родственный ей XML-элемент, когда используется внутри класса, отмеченного аннотацией `@Aspect`. Листинг 5.8 демонстрирует использование аннотации `@DeclareParents`.

Листинг 5.8. Внедрение интерфейса `Contestant` с использованием аннотаций `@AspectJ`

```
package com.springinaction.springidol;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
public class ContestantIntroducer {

    @DeclareParents(           // Внедрение интерфейса Contestant
        value = "com.springinaction.springidol.Performer",
        defaultImpl = GraciousContestant.class)
    public static Contestant contestant;
}
```

Как видите, `ContestantIntroducer` – это аспект. Но в отличие от аспектов, создававшихся до сих пор, он не содержит никаких советов. Зато он внедряет интерфейс `Contestant` в компоненты `Performer`. Подобно элементу `<aop:declare-parents>`, аннотация `@DeclareParents` состоит из трех частей.

- ❑ Атрибут `value` является эквивалентом атрибуту `types-matching` элемента `<aop:declare-parents>`. Он определяет тип компонентов, в которые должен быть внедрен интерфейс.
- ❑ Атрибут `defaultImpl` является эквивалентом атрибуту `defaultimpl` элемента `<aop:declare-parents>`. Он определяет класс, реализующий внедряемый интерфейс.
- ❑ Статическое свойство, к которому применяется аннотация `@DeclareParents`, определяет внедряемый интерфейс.

Как и для любых других аспектов, класс `ContestantIntroducer` следует объявить компонентом в контексте приложения Spring:

```
<bean class="com.springinaction.springidol.ContestantIntroducer" />
```

Теперь, когда элемент `<aop:aspectj-autoproxy>` обнаружит компонент, отмеченный аннотацией `@Aspect`, он автоматически создаст прокси-объект, делегирующий вызовы методов либо целевому объекту, либо реализации внедренного интерфейса, в зависимости от принадлежности вызываемого метода.

Обратите внимание, что аннотация `@DeclareParents` не имеет эквивалента атрибуту `delegate-ref` элемента `<aop:declare-parents>`. Это обусловлено тем, что `@DeclareParents` является аннотацией `@AspectJ`. `@AspectJ` – это независимый от Spring проект, поэтому его аннотации не обеспечивают поддержку компонентов. Отсюда следует, что если реализация внедряемого интерфейса оформлена в виде компонента, объявленного в конфигурации Spring, аннотация `@DeclareParents` может оказаться бесполезной и вам придется прибегнуть к использованию элемента `<aop:declare-parents>`.

Поддержка AOP в Spring позволяет отделять сквозные задачи от основной логики приложения. Но, как было показано выше, аспекты Spring все еще основаны на применении прокси-объектов, и круг их применения ограничен вызовами методов. Если вам потребуется нечто большее, чем применение советов к методам, подумайте о возможности использовать расширение AspectJ. В следующем разделе будет показано, как в приложениях на основе Spring использовать традиционные аспекты AspectJ.

5.5. Внедрение аспектов AspectJ

Возможностей Spring AOP вполне достаточно для многих случаев применения аспектов, но этот механизм является недостаточно мощным решением в сравнении с расширением AspectJ, предлагающим большее разнообразие типов точек сопряжения, отсутствующих в Spring AOP.

Например, точки сопряжения с конструкторами удобно использовать для применения советов к операциям создания объектов. В отличие от конструкторов в других объектно-ориентированных языках, конструкторы в языке Java отличаются от обычных методов, что не позволяет применять к ним аспекты Spring, основанные на прокси-объектах.



Аспекты расширения AspectJ независимы по отношению к Spring. Они могут быть вплетены в любое Java-приложение, включая приложения на основе Spring, но сам фреймворк при этом остается практически непричастен к ним.

Любой, хорошо спроектированный аспект, скорее всего, будет зависеть от других классов, играющих вспомогательную роль. Если выполнение совета аспекта зависит от одного или более классов, можно вручную создать их экземпляры вместе с аспектом. Или, еще лучше, использовать прием внедрения зависимостей для связывания компонентов с аспектами AspectJ.

Для иллюстрации создадим новый аспект в нашем примере «Spring Idol». В любом состязании должен участвовать судья, поэтому создадим аспект с использованием расширения AspectJ, представляющий судью, и назовем его JudgeAspect.

Листинг 5.9. Реализация судьи в конкурсе талантов в виде аспекта JudgeAspect

```
package com.springinaction.springidol;

public aspect JudgeAspect {
    public JudgeAspect() {}

    pointcut performance() : execution(* perform(..));

    after() returning() : performance() {
        System.out.println(criticismEngine.getCriticism());
    }

    // внедряется
    private CriticismEngine criticismEngine;

    public void setCriticismEngine(CriticismEngine criticismEngine) {
        this.criticismEngine = criticismEngine;
    }
}
```

Основная цель аспекта JudgeAspect – в том, чтобы дать комментарий к выступлению по его завершении. Срез множества точек сопряжения performance() в листинге 5.9 соответствует методу perform(). При объединении его с советом after() returning() получается аспект, реагирующий на завершение выступления.

Самое интересное, что можно заметить в листинге 5.9, – сам судья не дает собственных комментариев. Вместо этого аспект JudgeAspect использует вспомогательный объект CriticismEngine, вызывая его метод getCriticism(), возвращающий критический комментарий к выступлению. Чтобы избежать тесной связи между JudgeAspect и CriticismEngine, аспект JudgeAspect получает ссылку на объект CriticismEngine посредством внедрения зависимости через метод записи. Схематически эти отношения изображены на рис. 5.8.

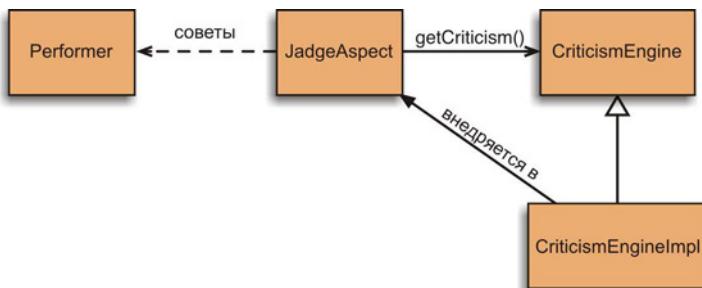


Рис. 5.8. Аспекты тоже должны внедряться.
Фреймворк Spring может внедрять аспекты AspectJ
в виде зависимостей, как если бы они были
обычными компонентами

CriticismEngine – это интерфейс, объявляющий метод getCriticism(). А его реализация в виде класса CriticismEngine представлена в листинге 5.10.

Листинг 5.10. Реализация интерфейса CriticismEngine, используемого аспектом JudgeAspect

```

package com.springinaction.springidol;

public class CriticismEngineImpl implements CriticismEngine {
    public CriticismEngineImpl() {}

    public String getCriticism() {
        int i = (int) (Math.random() * criticismPool.length);
        return criticismPool[i];
    }

    // внедряется
  
```

```
private String[] criticismPool;

public void setCriticismPool(String[] criticismPool) {
    this.criticismPool = criticismPool;
}

}
```

Класс CriticismEngineImpl реализует интерфейс CriticismEngine, выбирайая случайным образом комментарий из внедренного массива. Этот класс можно объявить компонентом Spring, как показано ниже:

```
<bean id="criticismEngine"
      class="com.springinaction.springidol.CriticismEngineImpl">
    <property name="criticisms">
        <list>
            <value>I'm not being rude, but that was appalling.</value>
            <value>You may be the least talented
                person in this show.</value>
            <value>Do everyone a favor and keep your day job.</value>
        </list>
    </property>
</bean>
```

Пока все идет неплохо. Теперь у нас есть реализация интерфейса CriticismEngine для передачи аспекту JudgeAspect. Осталось лишь внедрить объект CriticismEngineImpl в JudgeAspect.

Прежде чем я покажу, как выполнить внедрение, необходимо отметить, что аспекты AspectJ могут вплетаться в приложения вообще без привлечения фреймворка Spring. Однако если у вас появится желание использовать механизм внедрения зависимостей Spring, чтобы связать вспомогательный объект с аспектом AspectJ, этот аспект необходимо объявить компонентом в конфигурационном файле Spring. Следующее объявление компонента внедряет объект criticismEngine в JudgeAspect:

```
<bean class="com.springinaction.springidol.JudgeAspect"
      factory-method="aspectOf">
    <property name="criticismEngine" ref="criticismEngine" />
</bean>
```

По большей части данное объявление не особенно отличается от любых других, которые можно встретить в конфигурационных файлах Spring, разница – в использовании атрибута factory-method.

Экземпляры обычных компонентов Spring создаются контейнером Spring, но аспекты AspectJ создаются самим расширением AspectJ во время выполнения. К тому моменту, когда Spring получит возможность внедрить объект `CriticismEngine` в аспект `JudgeAspect`, последний уже будет создан.

Поскольку фреймворк Spring не участвует в создании аспекта `JudgeAspect`, недостаточно просто объявить его компонентом Spring. Нам нужно дать фреймворку возможность получить ссылку на экземпляр `JudgeAspect`, который уже создан расширением AspectJ, чтобы ее можно было связать с объектом `CriticismEngine`. Все аспекты AspectJ имеют статический метод `aspectOf()`, возвращающий один и тот же экземпляр аспекта. Поэтому, вместо того чтобы пытаться вызвать конструктор `JudgeAspect` для получения ссылки на экземпляр аспекта, необходимо задействовать атрибут `factory-method` для вызова метода `aspectOf()`.

Проще говоря, фреймворк Spring не использует объявление компонента, представленное выше, для создания экземпляра `JudgeAspect` – он создается расширением AspectJ во время выполнения. Вместо этого Spring получает ссылку на аспект вызовом фабричного метода `aspectOf()` и затем выполняет внедрение зависимости, как предусмотрено логикой работы элемента `<bean>`.

5.6. В заключение

AOP является мощным дополнением к объектно-ориентированному программированию. Аспекты позволяют группировать функциональные возможности в модули многократного использования, которые в противном случае оказались бы рассеяны по всему приложению. Вы можете точно указать, где и как должны использоваться эти функциональные возможности. Это позволяет избежать дублирования программного кода и при разработке прикладных классов сконцентрироваться на решении основной задачи.

Spring предоставляет фреймворк AOP, позволяющий окружать аспектами вызовы методов. Вы узнали, как вплетать советы до и/или после вызова метода, а также добавлять собственную обработку исключений.

Существуют несколько вариантов использования аспектов в приложениях на основе Spring. Определять советы и срезы множества точек сопряжения в Spring проще с помощью аннотаций `@AspectJ` и упрощенной схемы конфигурирования.



Наконец, бывают ситуации, когда для решения каких-либо задач возможностей Spring AOP оказывается недостаточно и приходится обращаться к расширению AspectJ для реализации более мощных аспектов. В таких ситуациях можно использовать Spring для внедрения зависимостей в аспекты AspectJ.

На данный момент мы познакомились с основами фреймворка Spring Framework. Было показано, как настроить контейнер Spring и как применять аспекты к объектам, управляемым фреймворком. Эти основные приемы позволяют создавать приложения, сконструированные из слабосвязанных объектов. В следующей главе рассматривается, как DI и AOP способствуют тестированию и как охватить программный код тестами.



Часть II. Основы приложений Spring

В первой части вы познакомились с контейнером Spring и имеющейся в нем поддержкой механизма внедрения зависимостей (DI) и аспектно-ориентированного программирования (AOP). Основываясь на этом фундаменте, во второй части будут исследоваться возможности фреймворка, которые предоставляются для создания корпоративных приложений.

Большинству приложений требуется хранить информацию в реляционных базах данных. В главе 6 «Работа с базами данных» будут представлены приемы использования поддержки баз данных в Spring для хранения данных. Здесь вы познакомитесь с поддержкой JDBC в Spring, которая поможет избавиться от массы шаблонного программного кода, обычно необходимого при работе с JDBC. Вы также увидите, как фреймворк Spring интегрируется с такими механизмами объектно-реляционного отображения, как Hibernate и JPA.

После освоения приемов хранения данных вам потребуется гарантировать их непротиворечивость. В главе 7 «Управление транзакциями» вы познакомитесь с возможностью декларативного применения транзакций к прикладным объектам с помощью AOP.

В главе 8 «Создание веб-приложений с помощью Spring MVC» вы познакомитесь с основами использования Spring MVC – веб-фреймворка, основанного на принципах Spring Framework. Здесь вы найдете множество контроллеров Spring MVC для обработки веб-запросов и увидите, как связывать параметры запросов с прикладными объектами, одновременно обеспечивая проверку данных и обработку ошибок.

Глава 9 «Использование Spring Web Flow» покажет, как конструировать диалоговые веб-приложения, предусматривающие выполнение операций пользователем в определенном порядке, с использованием фреймворка Spring Web Flow.

Безопасность – очень важный аспект большинства приложений, поэтому в главе 10 «Безопасность в Spring» будет показано, как с помощью фреймворка Spring Security защитить информацию, имеющуюся в приложении.



Глава 6. Работа с базами данных

В этой главе рассматриваются следующие темы:

- ❑ определение поддержки доступа к данным в Spring;
- ❑ конфигурирование баз данных;
- ❑ использование шаблонов Spring для работы с JDBC;
- ❑ совместное использование фреймворков Spring, Hibernate и JPA.

После знакомства с контейнером Spring пришло время применить полученные знания для создания действующего приложения. Лучше всего начать с требования, которое предъявляется к любому корпоративному приложению, такого как возможность хранения данных. Каждый из нас, вероятно, уже имел дело с *базами данных* в прошлом. При этом вы знаете, что доступ к данным имеет много препятствий. Нужно настроить фреймворк для доступа к данным, открыть соединения, обработать различные исключения и закрыть соединения. Если в этой последовательности что-то будет сделано неправильно, можно испортить или удалить важные данные. Если вам не приходилось испытывать последствия ошибок при обращении с данными, то поверьте на слово, это *весьма неприятно*.

Так как мы стремимся избежать неприятностей, обратимся за помощью к фреймворку Spring. В Spring имеется набор модулей для интеграции с различными технологиями хранения данных. Если для хранения данных используется непосредственно JDBC, iBATIS или фреймворк объектно-реляционного отображения (ORM), такой как Hibernate, Spring избавит вас от рутины при разработке программного кода, реализующего доступ к данным. Вместо возни с низкоуровневым доступом к данным можно положиться на Spring, который выполнит эту работу за вас, и сконцентрироваться на управлении данными в самом приложении.

В этой главе мы приступим к созданию Twitter-подобного приложения под названием Spitter. Это приложение станет основным примером для остальной части книги. Первое требование, которое мы постараемся удовлетворить, – реализовать слой, обеспечивающий хранение данных приложения Spitter.

Приступая к разработке этого слоя, мы сразу же сталкиваемся с необходимостью выбора. Для доступа к базам данных мы могли бы использовать JDBC, Hibernate, Java Persistence API (JPA) или любой другой подобный фреймворк. К счастью, Spring поддерживает все эти механизмы хранения данных, поэтому мы попробуем использовать каждый из них.

Но сначала заложим некоторые основы и познакомимся с философией хранения данных в Spring.

6.1. Философия доступа к данным в Spring

В предыдущих главах говорилось, что одна из целей Spring – обеспечить возможность разработки приложений, основанных на интерфейсах, в соответствии с принципами объектно-ориентированного программирования (ООП). Поддержка доступа к данным в Spring не является исключением.

Аббревиатура DAO¹ обозначает «объект доступа к данным» (Data Access Object), что прекрасно описывает роль DAO в приложении. Объекты доступа к данным являются средствами чтения и записи данных в базу данных. Они должны предоставлять эту функциональность через интерфейс, доступный остальной части приложения. Рисунок 6.1 демонстрирует правильный подход к проектированию *слоя доступа к данным*.

Как показано на рисунке, прикладные объекты получают доступ к DAO через интерфейсы. Такая организация дает несколько преимуществ. Во-первых, это упрощает тестирование прикладных объектов, так как они не связаны с конкретной реализацией доступа к данным. На самом деле можно создать фиктивные реализации этих интерфейсов доступа к данным, позволяющие проверять прикладные объекты без необходимости подключения к базе данных, что значительно ускоряет процесс тестирования и исключает вероятность ошибок из-за противоречивости данных.

¹ Многие разработчики, включая Мартина Фаулера (Martin Fowler), называют объекты доступа к данным *репозиториями*. Я высоко ценю рассуждения, которые ведут к использованию термина «репозиторий», но полагаю, что слово «репозиторий» имеет слишком широкое толкование и без этого дополнительного значения. Поэтому приношу свои извинения, но я не буду следовать этой популярной тенденции и продолжу называть эти объекты объектами доступа к данным (DAO).

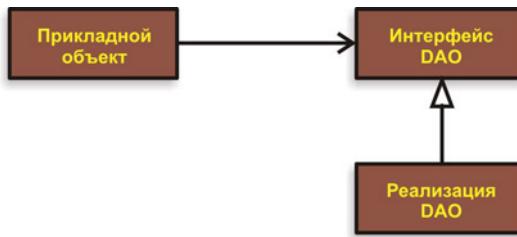


Рис. 6.1. Прикладные объекты не реализуют доступа к данным. Вместо этого они делегируют эти операции объектам доступа к данным. Интерфейс объектов DAO обеспечивает слабую связанныность с прикладными объектами

Кроме того, слой доступа к данным обеспечивает независимость от конкретной реализации хранилища. То есть подход на основе использования DAO обеспечивает «изоляцию» выбранного хранилища, предоставляя только ключевые методы доступа к данным через интерфейс. Это повышает гибкость архитектуры приложения и позволяет заменить используемый фреймворк на другой с минимальными воздействием на остальной программный код. Если детали реализации уровня доступа к данным будут просачиваться в другие части приложения, то все приложение окажется тесно связанным с уровнем доступа к данным, что сделает конструкцию приложения более жесткой.

Примечание. Если после прочтения двух предыдущих абзацев вы чувствуете, что я делаю сильный уклон в сторону скрытия слоя хранения данных за интерфейсами, то я рад, что смог донести до вас эту точку зрения. Я считаю, что интерфейсы являются ключом к написанию слабосвязанного кода и что они должны быть использованы во всех слоях приложения, а не только на уровне доступа к данным. Важно также отметить, что хотя фреймворк Spring поощряет использование интерфейсов, он не требует этого – вы можете использовать Spring для внедрения компонентов (объектов DAO или других) непосредственно в свойства других компонентов, не прибегая к использованию интерфейсов.

Один из способов, какими Spring может помочь оградить уровень доступа к данным от остальной части приложения, заключается в предоставлении последовательной иерархии исключений, используемых во всех модулях DAO.

6.1.1. Знакомство с иерархией исключений доступа к данным в Spring

Существует старый анекдот про парашютиста, которого снесло ветром на дерево, и он повис в нескольких метрах над землей. Спустя некоторое время появляется случайный прохожий, у которого незадачливый парашютист спрашивает, где он находится.

Прохожий: «Вы в шести метрах над землей».

Парашютист: «Вы, должно быть, программист?»

Прохожий: «Да, но как вы догадались?»

Парашютист: «Вы дали точный, но совершенно бессмысленный ответ».

Я слышал этот анекдот в разных вариациях, где прохожий имел разные профессии и национальности. Но он очень напоминает положение дел с исключением `SQLException` в JDBC. Если вам когда-нибудь приходилось работать с JDBC (без использования Spring), то вы наверняка хорошо знаете, что в JDBC и шагу нельзя ступить без необходимости перехватывать исключение `SQLException`. Это исключение свидетельствует, что произошла какая-то ошибка при обращении к базе данных, но оно содержит слишком мало информации о том, в чем, собственно, заключается ошибка и как ее обрабатывать.

В число распространенных проблем, которые могут вызывать исключение `SQLException`, входят:

- приложение не может подключиться к базе данных;
- выполняемый запрос имеет синтаксические ошибки;
- таблицы и/или столбцы, упомянутые в запросе, не существуют;
- попытка вставить или обновить значения нарушает ограничения базы данных.

Самый сложный вопрос, связанный с исключением `SQLException` определить, как правильно его обработать. Как оказывается, многие проблемы, вызвавшие исключение `SQLException`, не могут быть обработаны в блоке `catch`. Большинство исключений `SQLException` означают фатальную ошибку. Если приложение не может подключиться к базе данных, это обычно означает, что приложение не может продолжать работу. Аналогично, если ошибка содержится в запросе, трудно что-то сделать с этим во время выполнения.

Если после появления исключения `SQLException` ничего нельзя сделать для нормального продолжения работы приложения, зачем тогда его перехватывать?



Даже при наличии алгоритма решения некоторых проблем, связанных с исключением `SQLException`, вам все равно придется перехватить исключение `SQLException` и разбираться в его свойствах для получения дополнительной информации о характере проблемы. Это обусловлено тем, что исключение `SQLException` представляет собой «один ответ на все проблемы», связанные с доступом к данным. Вместо того чтобы иметь разные типы исключений для каждой возможной проблемы, используется исключение `SQLException`, которое возбуждается при любых проблемах доступа к данным.

Некоторые фреймворки доступа к базам данных предлагают более разнообразные иерархии исключений. `Hibernate`, например, предлагает около двух десятков различных исключений, каждое из которых соответствует определенной проблеме. Это делает возможным написать блоки `catch` для конкретных исключений, которые предполагается обрабатывать.

Но следует понимать, что исключения, реализованные в `Hibernate`, являются специфичными для этого фреймворка. А как было сказано выше, желательно было бы изолировать специфику механизма хранения в слое доступа к данным. Если в этом слое возбуждаются исключения `Hibernate`, то факт использования фреймворка `Hibernate` вне всяких сомнений просочится в остальные части приложения. Таким образом, или с этим придется смириться, или преобразовывать перехваченные исключения в некие новые, универсальные исключения, возбуждая их повторно.

С одной стороны, иерархия исключений `JDBC` является слишком общей. Фактически ее трудно даже назвать иерархией. С другой стороны, иерархия исключений в `Hibernate` является характерной для `Hibernate`. Нам же необходима иерархия исключений, достаточна информативная, но не связанная непосредственно с конкретным фреймворком доступа к данным.

Универсальная иерархия исключений в *Spring*

Модуль `Spring JDBC` реализует собственную иерархию исключений доступа к данным, позволяющую решить обе проблемы. В отличие от `JDBC`, в `Spring` имеются несколько исключений доступа к данным, каждое из которых описывает причины их возникновения. В табл. 6.1 приводится сравнение некоторых исключений доступа к данным в `Spring` и в `JDBC`.

Как видно из таблицы, исключения в `Spring` описывают практически все, что может произойти во время чтения или записи данных в базу. Кроме того, в табл. 6.1 перечислены далеко не все ис-

ключения доступа к данным, поддерживаемые фреймворком Spring. (Я мог бы перечислить их все, но мне не хотелось вызывать ощущение неполноценности JDBC.)

Хотя иерархия исключений в Spring гораздо богаче, чем простое исключение `SQLException` в JDBC, она не связана с каким-либо конкретным решением. Это означает, что фреймворк Spring будет возбуждать одни и те же исключения независимо от выбранного механизма доступа к данным. Это поможет изолировать выбранное решение в слое доступа к данным.

Таблица 6.1. Иерархия исключений JDBC в сравнении с исключениями доступа к данным в Spring

Исключения JDBC	Исключения доступа к данным в Spring
<code>BatchUpdateException</code>	<code>CannotAcquireLockException</code>
<code>DataTruncation</code>	<code>CannotSerializeTransactionException</code>
<code>SQLException</code>	<code>CleanupFailureDataAccessException</code>
<code>SQLWarning</code>	<code>ConcurrencyFailureException</code>
	<code>DataAccessException</code>
	<code>DataAccessResourceFailureException</code>
	<code>DataIntegrityViolationException</code>
	<code>DataRetrievalFailureException</code>
	<code>DeadlockLoserDataAccessException</code>
	<code>EmptyResultDataAccessException</code>
	<code>IncorrectResultSizeDataAccessException</code>
	<code>IncorrectUpdateSemanticsDataAccessException</code>
	<code>InvalidDataAccessApiUsageException</code>
	<code>InvalidDataAccessResourceUsageException</code>
	<code>OptimisticLockingFailureException</code>
	<code>PermissionDeniedDataAccessException</code>
	<code>PessimisticLockingFailureException</code>
	<code>TypeMismatchDataAccessException</code>
	<code>UncategorizedDataAccessException</code>

Смотрите! Мы избавились от блоков `catch!`

Из табл. 6.1 не очевидно, что все эти исключения наследуют исключение `DataAccessException`. Особенность исключения `DataAccessException` состоит в том, что оно является неконтролируемым исключением. Другими словами, нет необходимости перехватывать исключения доступа к данным, возбуждаемые фреймворком Spring (хотя при желании это возможно).

Исключение `DataAccessException` – это лишь частный пример многослойной философии Spring противопоставления контролируемых и неконтролируемых исключений. Фреймворк Spring придерживает-



ется позиции, согласно которой многие исключения являются результатом проблем, не разрешимых в блоке `catch`. Вместо того чтобы вынуждать разработчиков писать блоки обработки исключений (которые часто остаются пустыми), Spring продвигает идею неконтролируемых исключений. Это позволяет оставить решение о выборе места обработки исключений за разработчиком.

Чтобы воспользоваться преимуществами исключений доступа к данным в Spring, необходимо использовать один из шаблонов доступа к данным, поддерживаемых фреймворком. Посмотрим, как шаблоны Spring могут упростить работу с данными.

6.1.2. Шаблоны доступа к данным

Возможно, вам приходилось летать на самолете. Если это так, вы наверняка согласитесь, что одной из самых важных составляющих перелета является доставка вашего багажа из точки А в точку Б. Этот процесс включает в себя довольно много этапов. Когда вы приходите в аэропорт, ваша первая остановка будет у стойки проверки багажа. Далее служба безопасности проверит его для обеспечения безопасности полета. Затем его погрузят на тележку и перевезут к нужному самолету. Если вы летите с пересадкой с рейса на рейс, то и багаж должен быть перемещен вслед за вами. По прибытии в пункт назначения багаж должен быть извлечен из самолета и поставлен на транспортер. Наконец, вы спуститесь в зону получения багажа и заберете его.

Несмотря на большое количество этапов в этом процессе, вы активно участвуете лишь в паре из них. А перевозчик берет на себя ответственность за выполнение всех остальных необходимых действий. Лично вы участвуете лишь тогда, когда вам действительно нужно что-то сделать, об остальном же позаботятся без вашего участия. Это отражает сущность мощного шаблона проектирования: «Шаблонный метод» (Template Method).

Шаблонный метод определяет основу процесса. В нашем примере это процесс перемещения багажа из пункта вылета в пункт прибытия. Сам процесс фиксирован – он никогда не меняется. Общая последовательность обработки багажа протекает каждый раз одинаково: приемка багажа, погрузка в самолет и т. д. Некоторые этапы процесса также являются фиксированными – всякий раз выполняются одни и те же операции. По прибытии самолета в пункт назначения каждый предмет багажа выгружается и размещается на транспортере для выдачи багажа.

Однако в определенные моменты процесс делегирует свою работу подклассу, восполняющему некоторые особенности реализации. Это переменная часть процесса. Например, обработка багажа начинается с его проверки у стойки приема багажа. Эта часть процесса всегда должна следовать первой, так что его местоположение в процессе является фиксированным. Однако регистрация багажа для каждого пассажира отличается, и реализация этой части процесса определяется пассажиром. Если говорить в терминах программного обеспечения, шаблонный метод делегирует реализацию определенной части процесса интерфейсу. Различные реализации этого интерфейса определяют конкретные реализации данной части процесса.

И тот же шаблон применяется в Spring для доступа к данным. Независимо от используемых технологий для доступа к данным требуется выполнить определенные шаги. Например, всегда следует создавать подключение к хранилищу данных и освобождать ресурсы по завершении. Это – фиксированные шаги в процессе доступа к данным. Но каждый метод доступа к данным имеет свои отличительные особенности. Мы запрашиваем разные объекты и обновляем данные по-разному. Это – переменные шаги в процессе доступа к данным.

Фреймворк Spring разделяет фиксированные и переменные части процесса доступа к данным на два отдельных класса: *шаблоны* и *обратные вызовы*. Шаблоны управляют фиксированной частью процесса, а пользовательский программный код доступа к данным обслуживает обратные вызовы. Зоны ответственности этих двух классов изображены на рис. 6.2.

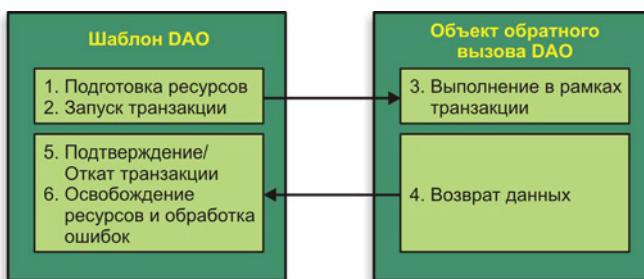


Рис. 6.2. Классы шаблонов DAO в Spring несут ответственность за общие функции доступа к данным. Для реализации специфических функций они могут обращаться к пользовательским объектам обратного вызова



Как показано на рис. 6.2, классы шаблонов в Spring заведуют фиксированными частями процесса доступа к данным – управление транзакциями, управление ресурсами и обработка исключений. А конкретные особенности доступа к данным, относящиеся к приложению, – создание отчетности, связывание параметров и передача результатов – обрабатываются реализацией обратного вызова. На практике такое разделение обеспечивает элегантное решение, поскольку вам достаточно побеспокоиться лишь о реализации своей части логики доступа к данным.

В Spring имеется несколько шаблонов на выбор, в зависимости от выбранного типа хранилища данных. При использовании JDBC лучшим выбором будет `JdbcTemplate`. А если вы предпочитаете один из фреймворков объектно-реляционного отображения, то более подходящим выбором будет `HibernateTemplate` или `JpaTemplate`. В табл. 6.2 перечислены все шаблоны для доступа к данным, имеющиеся в Spring, и их назначение.

Таблица 6.2. В составе Spring имеются несколько шаблонов для доступа к данным с использованием разных механизмов хранения

Класс шаблона (<code>org.springframework.*</code>)	Используется для взаимодействия с...
<code>jca.cci.core.CciTemplate</code>	Соединениями JCA CCI
<code>jdbc.core.JdbcTemplate</code>	Соединениями JDBC
<code>jdbc.core.namedparam.NamedParameterJdbcTemplate</code>	Соединениями JDBC, поддерживающими именованные параметры
<code>jdbc.core.simple.SimpleJdbcTemplate</code>	Соединениями JDBC, поддерживающими упрощенные конструкции Java 5
<code>orm.hibernate.HibernateTemplate</code>	Сеансами Hibernate 2.x
<code>orm.hibernate3.HibernateTemplate</code>	Сеансами Hibernate 3.x
<code>orm.ibatis.SqlMapClientTemplate</code>	Клиентами iBATIS SqlMap
<code>orm.jdo.JdoTemplate</code>	Реализациями Java Data Object
<code>orm.jpa.JpaTemplate</code>	Диспетчерами сущностей Java Persistence API

Как будет показано далее, для использования шаблона доступа к данным достаточно просто определить его в файле с настройками как обычный компонент в контексте Spring и затем связать с объектом DAO в приложении. Или же можно воспользоваться пре-

имуществом поддержки классов DAO в Spring для еще большего упрощения конфигурации приложения. Прямое связывание шаблонов – это хорошо, но Spring также предоставляет набор удобных базовых классов DAO, которые могут управлять шаблоном без вашего участия. Посмотрим, как действуют эти классы DAO, основанные на шаблонах.

6.1.3. Использование классов поддержки DAO

Шаблоны доступа к данным – это только часть модуля доступа к данным в Spring. Каждый шаблон также предоставляет удобные методы, упрощающие доступ к данным, избавляя от необходимости создавать явную реализацию обратного вызова. Кроме того, поверх конструкции «шаблон/обратный вызов» Spring предоставляет классы поддержки DAO для наследования вашими собственными классами DAO. Взаимосвязи между классом шаблона, классом поддержки DAO и вашей реализацией DAO схематически изображены на рис. 6.3.



Рис. 6.3. Отношения между прикладным объектом DAO, поддержкой DAO в Spring и классами шаблонов

Далее, когда будут исследованы варианты поддержки доступа к данным в Spring, будет показано, как классы поддержки DAO обеспечивают удобный доступ к поддерживаемым ими классам шаблонов. При создании прикладной реализации DAO можно создать свой класс, унаследовав в нем класс поддержки DAO, и вызвать метод получения шаблона, чтобы иметь прямой доступ к шаблону, лежащему в основе. Например, если прикладной объект DAO наследует класс поддержки `JdbcDaoSupport`, чтобы получить шаблон доступа `JdbcTemplate`, достаточно просто вызвать метод `getJdbcTemplate()`.

Плюс, если потребуется получить доступ непосредственно к фреймворку обслуживания хранилища, каждый из классов поддержки DAO обеспечивает доступ к любым классам, используемым для взаимодействия с базой данных. Например, класс `JdbcDaoSupport` содержит метод `getConnection()` для работы непосредственно с JDBC-соединением.

Подобно тому как фреймворк Spring предоставляет несколько реализаций классов шаблонов доступа к данным, он также обеспечивает несколько классов поддержки DAO, по одному для каждого шаблона. Классы поддержки DAO, входящие в состав Spring, перечислены в табл. 6.3.

Даже при том, что Spring обеспечивает поддержку лишь нескольких типов хранилищ, в этой главе не хватит места, чтобы охватить их все. Поэтому основное наше внимание будет сосредоточено на наиболее выгодных, с моей точки зрения, вариантах и тех, которые чаще используются на практике.

Таблица 6.3. Классы поддержки DAO обеспечивают удобный доступ к соответствующим шаблонам доступа к данным

Класс поддержки DAO (<code>org.springframework.*</code>)	Обеспечивает поддержку для...
<code>jca.cci.support.CciDaoSupport</code>	Соединений JCA CCI
<code>jdbc.core.support.JdbcDaoSupport</code>	Соединений JDBC
<code>jdbc.core.namedparam.NamedParameterJdbcDaoSupport</code>	Соединений JDBC, поддерживающих именованные параметры
<code>jdbc.core.simple.SimpleJdbcDaoSupport</code>	Соединений JDBC, поддерживающих упрощенные конструкции Java 5
<code>orm.hibernate.support.HibernateDaoSupport</code>	Сеансов Hibernate 2.x
<code>orm.hibernate3.support.HibernateDaoSupport</code>	Сеансов Hibernate 3.x
<code>orm.ibatis.support.SqlMapClientDaoSupport</code>	Клиентов iBATIS SqlMap
<code>orm.jdo.support.JdoDaoSupport</code>	Реализаций Java Data Object
<code>orm.jpa.support.JpaDaoSupport</code>	Диспетчеров сущностей Java Persistence API

Начнем с простого доступа через JDBC, так как это основной способ чтения и записи данных. Далее мы рассмотрим работу с Hibernate и JPA, двумя самыми популярными ORM-решениями на базе POJO.

Но обо всем по порядку – большинство вариантов поддержки хранилищ, имеющихся в Spring, будут зависеть от источника

данных. Поэтому, прежде чем приступить к созданию шаблонов и объектов DAO, необходимо настроить фреймворк Spring для работы с источником данных, чтобы обеспечить доступ объектов DAO к базе данных.

6.2. Настройка источника данных

Независимо от используемой формы поддержки DAO необходимо настроить ссылку на источник данных. Spring предлагает несколько вариантов настройки компонентов источников данных в приложении, в том числе:

- источники данных, определяемые драйвером JDBC;
- источники данных, найденные посредством JNDI;
- источники данных из пулов соединений.

Для промышленных приложений я рекомендую использовать источник данных, который берет соединения из пула соединений. Когда это возможно, я предпочитаю получать источник данных от сервера приложений через JNDI. Исходя из этого предпочтения, начнем изучение с того, как настроить Spring для извлечения источника данных из JNDI.

6.2.1. Использование источников данных из JNDI

Приложения на основе Spring обычно разворачиваются на сервере приложений Java EE, таком как WebSphere, JBoss или даже в веб-контейнере, таком как Tomcat. Эти серверы позволяют настраивать получение источников данных через JNDI. Преимущество настройки источников данных таким способом состоит в том, что ими можно полностью управлять вне приложения, оставив приложению только запрашивать доступ к источнику данных, когда оно будет готово к работе с базой данных. Более того, источники данных, управляемые сервером приложений, часто собираются в пулы для увеличения производительности и могут заменяться системными администраторами.

В Spring можно настроить ссылку на источник данных, хранимый в JNDI, и внедрять ее в классы при необходимости, как если бы он был обычным компонентом Spring. Элемент `<jee:jndi-lookup>` из пространства имен `jee` позволяет получить из JNDI любой объект, включая источники данных, и сделать его доступным как компонент

Spring. Например, если источник данных, необходимый приложению, находится в JNDI, его можно внедрить в Spring с помощью элемента `<jee:jndi-lookup>`, как показано ниже:

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true" />
```

Атрибут `jndi-name` определяет имя ресурса в JNDI. Если определен только атрибут `jndi-name`, будет выполнен поиск источника данных по указанному имени. Но если приложение выполняется внутри сервера приложений Java, тогда можно дополнительно присвоить атрибуту `resource-ref` значение `true`, благодаря чему имя в атрибуте `jndi-name` будет дополнено слева приставкой `java:comp/env/`.

6.2.2. Использование пулов соединений

Если не удалось получить источник данных из JNDI, следующим наилучшим выходом является настройка пулов соединений непосредственно в Spring. Хотя фреймворк Spring не предоставляет собственного пула, подходящая реализация имеется в проекте Jakarta Commons Database Connection Pooling (DBCP) (<http://jakarta.apache.org/commons/dbcp>).

DBCP включает несколько источников данных, предоставляющих пулы соединений, но обычно используется класс `BasicDataSource`, потому что он прост в настройке и напоминает класс `DriverManagerDataSource` в Spring (о котором мы будем рассказывать далее).

Для приложения Spitter компонент типа `BasicDataSource` настраивается следующим образом:

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
    <property name="url"
        value="jdbc:hsqldb:hsqldb://localhost/spitter/spitter" />
    <property name="username" value="sa" />
    <property name="password" value="" />
    <property name="initialSize" value="5" />
    <property name="maxActive" value="10" />
</bean>
```

Первые четыре свойства – это начальные настройки `BasicDataSource`. Свойство `driverClassName` определяет полное имя класса драйвера JDBC. Здесь выполняются настройки драйвера JDBC для базы данных Hypersonic. Свойство `url` определяет полный JDBC URL базы данных. Наконец, свойства `username` и `password` определяют параметры аутентификации при подключении к базе данных.

Эти четыре основных свойства определяют сведения о подключении для `BasicDataSource`. Кроме того, некоторые свойства могут быть использованы для настройки самого пула источников данных. В табл. 6.4 перечислены некоторые из наиболее полезных свойств конфигурации пула для `BasicDataSource`.

Для наших целей мы настроили пул с начальной емкостью в пять соединений. Если будет необходимо больше одновременных соединений, `BasicDataSource` будет добавлять их вплоть до установленного максимума в 10 активных соединений.

Таблица 6.4. Свойства конфигурации пула `BasicDataSource`

Свойство конфигурации пула	Назначение
<code>initialSize</code>	Начальное число соединений при создании пула
<code>maxActive</code>	Максимально допустимое число одновременно открытых активных соединений. Значение 0 соответствует неограниченному числу соединений
<code>maxIdle</code>	Максимально допустимое число простояющих соединений, которые не будут закрыты. Значение 0 соответствует неограниченному числу соединений
<code>maxOpenPreparedStatements</code>	Максимально допустимое количество скомпилированных запросов, которые могут быть помещены в пул запросов одновременно. Значение 0 соответствует неограниченному числу запросов
<code>maxWait</code>	Время ожидания пулом возврата соединения в пул (при отсутствии свободных соединений) до возбуждения исключения. Значение <code>-1</code> соответствует бесконечному ожиданию
<code>minEvictableIdleTimeMillis</code>	Максимальное время простоя соединения, прежде чем оно может быть удалено из пула
<code>minIdle</code>	Минимальное число простояющих соединений, которые могут оставаться в пуле без создания новых соединений
<code>poolPreparedStatements</code>	Признак поддержки пула скомпилированных запросов (логическое значение)



6.2.3. Источник данных JDBC

Простейшие источники данных, которые только можно настроить в Spring, – это те, что определены с использованием драйвера JDBC. Spring предлагает на выбор два класса таких источников данных (оба в пакете `org.springframework.jdbc.datasource`).

- ❑ `DriverManagerDataSource` – каждый раз, когда запрашивается соединение, возвращает новое соединение. В отличие от `BasicDataSource` в DBCP, соединения, предоставляемые `DriverManagerDataSource`, не объединяются в пул.
- ❑ `SingleConnectionDataSource` – каждый раз, когда запрашивается соединение, возвращает одно и то же соединение. Хотя `SingleConnectionDataSource` и не является пулом в полном смысле этого слова, тем не менее его можно воспринимать как источник данных с пулом, содержащим единственное соединение.

Настройка любого из этих источников данных напоминает настройку `BasicDataSource` в DBCP:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.
      DriverManagerDataSource">
    <property name="driverClassName"
              value="org.hsqldb.jdbcDriver" />
    <property name="url"
              value="jdbc:hsqldb:hsq://localhost/spitter/spitter" />
    <property name="username" value="sa" />
    <property name="password" value="" />
</bean>
```

Разница лишь в том, что поскольку ни `DriverManagerDataSource`, ни `SingleConnectionDataSource` не предоставляют пулов соединений, в них нет свойств настройки пула.

Классы `SingleConnectionDataSource` и `DriverManagerDataSource` прекрасно подходят для небольших приложений и для использования в процессе разработки, тем не менее следует серьезно рассматривать последствия использования любого из них в действующем приложении. Поскольку класс `SingleConnectionDataSource` имеет одно и только одно соединение для работы с базой данных, он плохо подходит для использования в многопоточных приложениях. В то же время, несмотря на то что класс `DriverManagerDataSource` способен поддерживать несколько потоков, при каждом запросе на соединение он несет по-

тери производительности на создание этого соединения. Из-за этих ограничений я настоятельно рекомендую использовать в качестве источников данных пулы соединений.

Теперь, когда мы установили соединение с базой данных через источник данных, можно приступать к фактической работе с базой данных. Как уже упоминалось, фреймворк Spring предлагает несколько вариантов выполнения операций с базами данных, включая JDBC, Hibernate и Java Persistence API (JPA). В следующем разделе рассматривается порядок создания уровня доступа к данным в приложениях на основе Spring с использованием поддержки JDBC, предлагаемой фреймворком. Но если вас больше интересует использование Hibernate или JPA, тогда вы можете сразу перейти к разделу 6.4 или 6.5 соответственно.

6.3. Использование JDBC совместно со Spring

Существует множество технологий хранения данных. Hibernate, iBATIS и JPA – лишь некоторые из них. Несмотря на немалое количество вариантов, записывать Java-объекты прямо в базу данных – это уже немного старомодный путь для заработка. Стоп, а как же люди теперь зарабатывают деньги?! А, проверенным дедовским методом – сохраняя данные с помощью старого доброго JDBC.

А почему бы и нет? JDBC не требует владения языком запросов другого фреймворка. Он основан на SQL – языке доступа к данным. Плюс при использовании JDBC можно куда более точно настроить производительность доступа к данным в сравнении с любыми другими технологиями. И JDBC позволяет пользоваться всеми преимуществами конкретных особенностей базы данных, тогда как другие фреймворки могут препятствовать этому или даже запрещать.

Более того, JDBC дает возможность работать с данными на гораздо более низком уровне, чем прочие фреймворки доступа к данным, позволяя, например, манипулировать отдельными столбцами в базе данных. Такие широкие возможности доступа к данным могут пригодиться, например, в приложениях создания отчетов, где нет смысла преобразовывать данные в объекты, чтобы затем вновь извлекать эти данные из объектов, преобразуя их практически к исходному виду.

Но не все так безоблачно в мире JDBC. К его мощности, гибкости, и простоте прилагаются некоторые недостатки.



6.3.1. Борьба с разбуханием JDBC-кода

Несмотря на то что JDBC API действует в тесном контакте с базой данных, ответственность за управление всем, что касается доступа к базе данных, возлагается на программиста. Сюда входят: управление ресурсами базы данных и обработка исключений.

Если вам приходилось писать программы, использующие JDBC для вставки данных в базу, фрагмент, представленный в листинге 6.1, не покажется вам чем-то необычным.

Листинг 6.1. Использование JDBC для вставки строки в базу данных

```
private static final String SQL_INSERT_SPITTER =
    "insert into spitter (username, password, fullname) values (?, ?, ?)";
private DataSource dataSource;
public void addSpitter(Spitter spitter) {
    Connection conn = null;
    PreparedStatement stmt = null;
    try {
        conn = dataSource.getConnection(); // Получить соединение

        stmt = conn.prepareStatement(SQL_INSERT_SPITTER); // Создать запрос

        stmt.setString(1, spitter.getUsername()); // Связать параметры
        stmt.setString(2, spitter.getPassword());
        stmt.setString(3, spitter.getFullName());

        stmt.execute(); // Выполнить запрос

    } catch (SQLException e) { // Обработать исключение (как-нибудь)
        // выполнить что-нибудь... хотя... не уверен, что тут можно сделать
    } finally {
        try {
            if (stmt != null) { // Освободить ресурсы
                stmt.close();
            }
            if (conn != null) {
                conn.close();
            }
        } catch (SQLException e) {
            // Я еще менее уверен, что тут можно сделать
        }
    }
}
```

Святые угодники! Более 20 строк кода, чтобы вставить простой объект в базу данных. При этом сами операции JDBC выглядят проще некуда. Неужели обязательно писать так много строк, чтобы выполнить простейшую операцию? Конечно же нет. В действительности здесь лишь несколько строк реализуют вставку. Но JDBC требует корректной обработки соединений и запросов, а также возможного исключения SQLException.

Что касается исключения SQLException: беда в том, что здесь совершенно не понятно, как его обрабатывать (поскольку неочевидны проблемы, вызвавшие его), но вы вынуждены дважды предусматривать его обработку! Его следует перехватить на случай, если ошибка возникла в момент вставки новой записи, и снова перехватить его при закрытии запроса и соединения. Слишком сложно, особенно если учесть, что некоторые ошибки не могут быть обработаны программно.

Теперь посмотрим на фрагмент в листинге 6.2, выполняющий обновление строки в таблице Spitter с применением традиционного подхода к использованию JDBC.

Листинг 6.2. Использование JDBC для обновления строки в базе данных

```
private static final String SQL_UPDATE_SPITTER =
        "update spitter set username = ?, password = ?, fullname = ?"
        + "where id = ?";
public void saveSpitter(Spitter spitter) {
    Connection conn = null;
    PreparedStatement stmt = null;
    try {
        conn = dataSource.getConnection(); // Получить соединение

        stmt = conn.prepareStatement(SQL_UPDATE_SPITTER); // Создать запрос

        stmt.setString(1, spitter.getUsername()); // Связать параметры
        stmt.setString(2, spitter.getPassword());
        stmt.setString(3, spitter.getFullName());
        stmt.setLong(4, spitter.getId());

        stmt.execute(); // Выполнить запрос
    } catch (SQLException e) { // Обработать исключение (как-нибудь)
        // Все еще не уверен, что можно было бы предложить здесь
    } finally {
        try {
            if (stmt != null) { // Освободить ресурсы
                stmt.close();
            }
        }
```

```
        }
        if (conn != null) {
            conn.close();
        }
    } catch (SQLException e) {
    // и здесь
}
}
```

На первый взгляд, листинг 6.2 может показаться идентичным листингу 6.1. На самом деле, не считая строку с SQL-инструкцией и строку, где создается сам запрос, они идентичны. Опять же, получился большой объем кода, чтобы выполнить простую операцию по изменению одной строки в базе данных. Более того, здесь существует масса повторяющегося кода. В идеале следовало бы написать строки, характерные для конкретной задачи. В конце концов, листинг 6.2 отличается от листинга 6.1 лишь несколькими строками. Все остальное – это шаблонный код.

В заключение обзора использования JDBC посмотрим, как можно было бы реализовать извлечение данных из базы. Как показано в листинге 6.3, реализация этой операции также не отличается красотой.

Листинг 6.3. Использование JDBC для извлечения строки из базы данных

```
private static final String SQL_SELECT_SPITTER =
    "select id, username, fullname from spitter where id = ?";
public Spitter getSpitterById(long id) {
    Connection conn = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        conn = dataSource.getConnection(); // Получить соединение
        stmt = conn.prepareStatement(SQL_SELECT_SPITTER); // Создать запрос
        stmt.setLong(1, id); // Связать параметры
        rs = stmt.executeQuery(); // Выполнить запрос
        Spitter spitter = null;
```

```
if (rs.next()) { // Обработать результаты
    spitter = new Spitter();
    spitter.setId(rs.getLong("id"));
    spitter.setUsername(rs.getString("username"));
    spitter.setPassword(rs.getString("password"));
    spitter.setFullName(rs.getString("fullname"));
}
return spitter;
} catch (SQLException e) { // Обработать исключение (как-нибудь)

} finally {
    if(rs != null) {
        try { // Освободить ресурсы
            rs.close();
        } catch(SQLException e) {}
    }
    if(stmt != null) {
        try {
            stmt.close();
        } catch(SQLException e) {}
    }
    if(conn != null) {
        try {
            conn.close();
        } catch(SQLException e) {}
    }
}
return null;
}
```

Данный пример содержит такую же массу шаблонного кода, как и предыдущие примеры, и даже больше. Это словно *принцип Парето*¹, перевернутый с ног на голову: чтобы выполнить запрос-строку, необходимо написать 20 процентов программного кода, тогда как оставшиеся 80 процентов – просто шаблонный код.

К настоящему моменту вы уже должны понять, что при использовании JDBC приходится писать массу шаблонного кода для создания соединений, запросов и обработки исключений. Начиная с этого места, я закончу, наконец, эту пытку и не буду заставлять вас дальше смотреть на этот ужасный код.

¹ http://ru.wikipedia.org/wiki/Закон_Парето.



Хотя в действительности этот шаблонный код играет важную роль. Освобождение ресурсов и обработки ошибок делают доступ к данным надежнее. Без этого ошибки останутся незамеченными, а ресурсы – открытыми, что может привести к непредсказуемым последствиям и утечке ресурсов. Поэтому мало того, что этот код необходим, мы также должны убедиться, что он не содержит ошибок. Это еще одна причина, чтобы позволить фреймворку самому иметь дело с этим шаблонным кодом, так как мы знаем, что он написан один раз и написан правильно.

6.3.2. Работа с шаблонами JDBC

Модуль JDBC в Spring освобождает от необходимости управления ресурсами и обработки исключений. Он дает свободу писать только тот код, который необходим для перемещения данных в базу данных и обратно.

Как говорилось выше, в разделе 6.3.1, фреймворк Spring скрывает весь вспомогательный код доступа к данным за классами шаблонов. Для работы с JDBC фреймворк Spring предоставляет три класса шаблонов, на выбор:

- ❑ `JdbcTemplate` – самый основной шаблон JDBC, этот класс предоставляет простой доступ к базе данных через JDBC и простые запросы с индексированными параметрами;
- ❑ `NamedParameterJdbcTemplate` – этот шаблон JDBC позволяет создавать запросы с именованными параметрами вместо индексированных;
- ❑ `SimpleJdbcTemplate` – эта версия шаблона JDBC использует новые возможности Java 5, такие как автоматическая упаковка и распаковка (`autoboxing`), шаблонные классы (`generics`) и списки аргументов переменной длины (`varargs`) для упрощения работы с шаблоном.

В прошлом приходилось тщательно взвешивать выбор того или иного шаблона JDBC. Но в последних версиях Spring сделать выбор намного проще. В версии Spring 2.5 поддержка именованных параметров из `NamedParameterJdbcTemplate` была добавлена в `SimpleJdbcTemplate`. А в версии Spring 3.0 была ликвидирована поддержка старых версий Java (ниже версии Java 5), поэтому выбор шаблона `JdbcTemplate` не дает никаких преимуществ перед `SimpleJdbcTemplate`. В свете этих изменений в данной главе мы сосредоточимся исключительно на использовании `SimpleJdbcTemplate`.

Доступ к данным с использованием SimpleJdbcTemplate

Для выполнения своей работы шаблону SimpleJdbcTemplate необходим только компонент, реализующий интерфейс DataSource. Это делает компонент типа SimpleJdbcTemplate достаточно простым в настройке, как показано ниже:

```
<bean id="jdbcTemplate"
      class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
    <constructor-arg ref="dataSource" />
</bean>
```

Фактически компонентом типа DataSource, на который ссылается свойство dataSource, может быть любая реализация javax.sql.DataSource, включая те, что представлены в разделе 6.2.

Теперь можно внедрить SimpleJdbcTemplate в наш объект DAO и использовать его для доступа к базе данных. Например, предположим, что Spitter DAO основан на SimpleJdbcTemplate:

```
public class JdbcSpitterDAO implements SpitterDAO {
    ...
    private SimpleJdbcTemplate jdbcTemplate;
    public void setJdbcTemplate(SimpleJdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }
}
```

Тогда свойство jdbcTemplate компонента JdbcSpitterDAO можно было бы связать, как показано ниже:

```
<bean id="spitterDao"
      class="com.habuma.spitter.persistence.SimpleJdbcTemplateSpitterDao">
    <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

Передав в распоряжение нашего объекта DAO шаблон SimpleJdbcTemplate, можно существенно упростить метод addSpitter() из листинга 6.1. Новый метод addSpitter(), основанный на применении шаблона SimpleJdbcTemplate, представлен в листинге 6.4.

Листинг 6.4. Метод addSpitter(), основанный на применении шаблона SimpleJdbcTemplate

```
public void addSpitter(Spitter spitter) {  
    jdbcTemplate.update(SQL_INSERT_SPITTER, // Добавить запись  
        spitter.getUsername(),  
        spitter.getPassword(),  
        spitter.getFullName(),  
        spitter.getEmail(),  
        spitter.isUpdateByEmail());  
    spitter.setId(queryForIdentity());  
}
```

Думаю, вы согласитесь, что эта версия метода `addSpitter()` выглядит значительно проще. Здесь отсутствует программный код, создающий соединение и запрос, а также отсутствует обработка исключений. Нет ничего, кроме чистого кода вставки данных.

Только потому, что вы не видите здесь шаблонного кода, это не означает, что его нет на самом деле. Он просто грамотно спрятан внутри класса шаблона. Когда вызывается метод `update()`, `SimpleJdbcTemplate` автоматически получает соединение, создает запрос и выполняет его.

Здесь также не видно, как обрабатываются исключения `SQLException`. Внутри класс `SimpleJdbcTemplate` перехватывает все исключения `SQLException` и преобразует обобщенное исключение `SQLException` в одно из более конкретных исключений доступа к данным, перечисленных в табл. 6.1, и повторно возбуждает их. Так как в Spring все исключения доступа к данным являются исключениями времени выполнения, их необязательно перехватывать в методе `addSpitter()`.

Чтение данных с использованием `SimpleJdbcTemplate` также упрощается. В листинге 6.5 представлена новая версия `getSpitterById()`, где используются обратные вызовы `SimpleJdbcTemplate` для отображения возвращаемого набора данных в объекты предметной области.

Листинг 6.5. Выборка данных с помощью SimpleJdbcTemplate

```
public Spitter getSpitterById(long id) {  
    return jdbcTemplate.queryForObject( // Запрос на получение данных  
        SQL_SELECT_SPITTER_BY_ID,  
        new ParameterizedRowMapper<Spitter>() {  
            public Spitter mapRow(ResultSet rs, int rowNum)  
                throws SQLException {  
                Spitter spitter = new Spitter(); // Отображение
```

```
        spitter.setId(rs.getLong(1));           // результатов
        spitter.setUsername(rs.getString(2));    // в объект
        spitter.setPassword(rs.getString(3));
        spitter.setFullName(rs.getString(4));
        return spitter;
    }
},
id
);
}
}
```

Этот метод `getSpitterById()` использует для запроса данных метод `queryForObject()` класса `SimpleJdbcTemplate`. Метод `queryForObject()` принимает три параметра:

- ❑ значение типа `String`, содержащее строку SQL-запроса для выборки данных;
- ❑ объект `ParameterizedRowMapper`, извлекающий значения из объекта `ResultSet` и конструирующий объект предметной области (в данном случае – объект `Spitter`);
- ❑ список аргументов переменной длины со значениями для связывания с индексированными параметрами запроса.

Настоящее волшебство творится в объекте `ParameterizedRowMapper`. Для каждой строки в наборе данных, возвращаемом запросом, `SimpleJdbcTemplate` будет вызывать метод `mapRow()` объекта `RowMapper`. Внутри `ParameterizedRowMapper` мы добавили код, создающий объект `Spitter` и заполняющий его свойства значениями из `ResultSet`.

Подобно методу `addSpitter()`, метод `getSpitterById()` не содержит шаблонного кода. В отличие от традиционного использования JDBC, здесь отсутствует код, реализующий управление ресурсами или обработку исключений. Методы, использующие шаблон `SimpleJdbcTemplate`, сфокусированы исключительно на извлечении объектов `Spitter` из базы данных.

Использование именованных параметров

Метод `addSpitter()`, представленный в листинге 6.4, использует индексированные параметры. Это означает необходимость обращать внимание на порядок следования параметров в запросе и при вызове метода `update()` передавать их значения в таком же порядке. Если позднее потребуется изменить SQL-запрос так, что при этом изменится порядок следования параметров, нам также придется привести в соответствие список значений, передаваемых методу.



При желании можно было бы использовать только именованные параметры. Они позволяют присвоить имя каждому параметру в SQL-запросе и впоследствии обращаться к параметрам по их именам, реализуя связывание передаваемых значений. Например, рассмотрим запрос `SQL_INSERT_SPITTER`, объявленный следующим образом:

```
private static final String SQL_INSERT_SPITTER =  
    "insert into spitter (username, password, fullname) " +  
    "values (:username, :password, :fullname)";
```

При использовании именованных параметров порядок их следования при связывании значений не имеет значения. Каждое значение можно связать по имени параметра. После изменения SQL-запроса, если при этом изменится порядок их следования, программный код их подстановки изменять не потребуется.

Для использования именованных параметров в Spring 2.0 необходимо было использовать специальный класс шаблона `NamedParameterJdbcTemplate`. А в более ранних версиях Spring это вообще было невозможно. Но в версии Spring 2.5 поддержка именованных параметров, реализованная в шаблоне `NamedParameterJdbcTemplate`, была добавлена в шаблон `SimpleJdbcTemplate`, поэтому у нас уже имеется все необходимое, чтобы задействовать именованные параметры в методе `addSpitter()`. В листинге 6.6 приводится новая версия `addSpitter()`, использующая именованные параметры.

Листинг 6.6. Использование именованных параметров при работе с шаблонами JDBC в Spring

```
public void addSpitter(Spitter spitter) {  
    Map<String, Object> params = new HashMap<String, Object>();  
    params.put("username", spitter.getUsername()); // Связывание параметров  
    params.put("password", spitter.getPassword());  
    params.put("fullname", spitter.getFullName());  
  
    jdbcTemplate.update(SQL_INSERT_SPITTER, params); // Вставка  
    spitter.setId(queryForIdentity());  
}
```

Первое, на что следует обратить внимание в этой версии метода `addSpitter()`, – эта версия получилась немного длиннее предыдущей. Это обусловлено тем, что связывание именованных параметров про-

изводится посредством `java.util.Map`. Несмотря на это, каждая строка подчинена главной цели – вставке объекта `Spitter` в базу данных. Здесь также отсутствует код управления ресурсами или обработки исключений, загромождающий главную цель метода.

Использование классов поддержки DAO при работе с JDBC

При создании любых прикладных классов, основанных на JDBC DAO, следует обязательно добавлять свойство типа `SimpleJdbcTemplate` и соответствующий ему метод записи. А затем связывать компонент `SimpleJdbcTemplate` со свойством типа `SimpleJdbcTemplate` каждого объекта DAO. Это несложно, когда приложение содержит единственный объект DAO, но при большом их количестве потребуется писать массу повторяющегося кода.

Для решения этой проблемы можно было бы создать общий родительский класс для всех объектов DAO и разместить в нем свойство `SimpleJdbcTemplate`. После этого во всех классах DAO можно было бы просто наследовать этот класс и использовать для доступа к данным свойство `SimpleJdbcTemplate` родительского класса. Рисунок 6.4 иллюстрирует предлагаемую схему отношений между прикладными и базовым классами DAO.

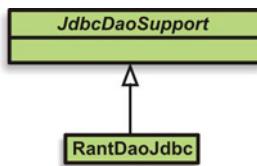


Рис. 6.4. Классы поддержки DAO в Spring предусматривают свойство для хранения объектов шаблонов JDBC, чтобы их подклассам не требовалось создавать собственные экземпляры шаблонов JDBC

Идея создания базового класса DAO, который включал бы в себя шаблон JDBC, выглядит весьма привлекательно, особенно если учесть, что фреймворк Spring уже содержит такой базовый класс. В действительности фреймворк содержит три таких класса: `JdbcDaoSupport`, `SimpleJdbcDaoSupport` и `NamedParameterJdbcDaoSupport` – по одному на каждый шаблон JDBC в Spring. Чтобы задействовать любой из этих классов поддержки DAO, достаточно наследовать свой класс от него. Например:

```
public class JdbcSpitterDao extends SimpleJdbcDaoSupport
    implements SpitterDao {
...
}
```

Класс `SimpleJdbcDaoSupport` обеспечивает удобный доступ к шаблону `SimpleJdbcTemplate` через метод `getJdbcTemplate()`. Например, метод `addSpitter()` можно было бы переписать, как показано ниже:

```
public void addSpitter(Spitter spitter) {
    getSimpleJdbcTemplate().update(SQL_INSERT_SPITTER,
        spitter.getUsername(),
        spitter.getPassword(),
        spitter.getFullName(),
        spitter.getEmail(),
        spitter.isUpdateByEmail());
    spitter.setId(queryForIdentity());
}
```

При настройке класса DAO в Spring можно непосредственно связать компонент `SimpleJdbcTemplate` со свойством `jdbcTemplate`, как показано ниже:

```
<bean id="spitterDao"
      class="com.habuma.spitter.persistence.JdbcSpitterDao">
    <property name="jdbcTemplate" ref="jdbcTemplate" />
</bean>
```

Эта настройка будет работать, но она мало чем отличается от прежней настройки объекта DAO, не наследующего класс `SimpleJdbcDaoSupport`. При желании можно опустить компонент-посредник и связать источник данных непосредственно со свойством `dataSource`, которое класс `JdbcSpitterDao` наследует от класса `SimpleJdbcDaoSupport`:

```
<bean id="spitterDao"
      class="com.habuma.spitter.persistence.JdbcSpitterDao">
    <property name="dataSource" ref="dataSource" />
</bean>
```

При настройке свойства `dataSource` компонента `JdbcSpitterDao` нев явно создается экземпляр класса `SimpleJdbcTemplate`, что избавляет от необходимости явно объявлять компонент `SimpleJdbcTemplate` в конфигурации Spring.

JDBC – это самый низкоуровневый способ доступа к данным в реляционных базах данных. Шаблоны JDBC в Spring избавляют от мучений писать шаблонный код управления ресурсами соединений и обработки исключений, давая возможность сосредоточиться на основной работе по извлечению, сохранению и обновлению данных.

Даже при том, что Spring снимает большую часть головной боли при работе с JDBC, с увеличением размера и сложности приложения этот путь все равно может стать слишком сложным. Чтобы упростить решение проблем доступа к данным в больших приложениях, можно задействовать такие фреймворки доступа к базам данных, как Hibernate. Посмотрим, как задействовать Hibernate в слое хранения данных приложения на основе Spring.

6.4. Интеграция Hibernate и Spring

Когда мы были детьми, езда на велосипеде была забавой, не так ли? С утра мы ехали в школу. После уроков отправлялись домой к лучшему другу. Когда же становилось поздно и родители начинали звать нас, ругаясь, что мы шляемся дотемна, мы стремглав мчались домой по темноте. Это были золотые деньги.

Затем, когда мы выросли, велосипеда стало слишком мало. Кому-то приходится преодолевать длинный путь от дома до работы. Нужно завезти продукты домой и отвезти детей на тренировку по футболу. А если вы еще и живете в Техасе, то без кондиционера это уже не жизнь! В общем, наши потребности просто переросли велосипед.

JDBC – это велосипед в мире технологий хранения данных. Он отлично подходит, для чего придуман, и в некоторых ситуациях просто великолепен. Но поскольку наши приложения становятся все более сложными, попробуем сформулировать наши новые требования к хранению данных. Итак, мы должны иметь возможность отображать свойства объектов в столбцы базы данных, и было бы неплохо, если бы SQL-запросы создавались автоматически, без необходимости вручную вводить бесконечные строки из вопросительных знаков. Нам также нужны некоторые, более сложные «фишки».

- ❑ *Отложенная загрузка* (*lazy loading*) – иерархии наших объектов становятся все более сложными, поэтому иногда бывает нежелательно извлекать все дерево наследования немедленно. Например, представьте, что приложение выбирает коллекцию объектов `PurchaseOrder`, где каждый из этих объектов, в свою очередь, содержит коллекцию объектов `LineItem`. Если в дан-



ный момент приложению требуется получить только атрибуты объектов `PurchaseOrder`, нет смысла извлекать данные для объектов `LineItem`. Это весьма накладно. Отложенная загрузка позволяет извлекать только те данные, которые действительно необходимы.

- **Полная загрузка (eager fetching)** – полная противоположность отложенной загрузке. Полная загрузка позволяет извлечь все дерево объектов в одном запросе. Когда точно известно, что приложению потребуются все объекты `PurchaseOrder` и в связанные с ними объекты `LineItems`, полная загрузка позволит получить их из базы данных за одну операцию, избавляя от накладных расходов на повторные обращения к ней.
- **Каскадирование (cascading)** – иногда изменения в одной таблице базы данных должны привести к изменениям в других таблицах. Возвращаясь к примеру с объектами `PurchaseOrder`: при удалении объекта `PurchaseOrder` обычно бывает желательно удалить из базы данных все связанные с ним объекты `LineItems`.

Существуют фреймворки, способные предоставить подобную функциональность. Общее название для такой функциональности – *объектно-реляционное отображение* (Object-Relational Mapping, ORM). Применение инструмента ORM для слоя, реализующего хранение данных, может буквально спасти от необходимости писать тысячи строк лишнего кода и сэкономить массу времени на его разработку. Это позволит сместить фокус с написания чреватого ошибками SQL-кода на решение прикладных задач самого приложения.

Фреймворк Spring включает поддержку некоторых фреймворков, реализующих хранение данных, включая Hibernate, iBATIS, Java Data Objects (JDO) и Java Persistence API (JPA).

Как и в случае с поддержкой JDBC, Spring обеспечивает поддержку фреймворков ORM, предоставляя точки интеграции для фреймворков, а также некоторые дополнительные услуги, такие как:

- интегрированная поддержка декларативных транзакций;
- прозрачная обработка исключений;
- легковесные классы шаблонов с поддержкой выполнения в многопоточной среде;
- классы поддержки DAO;
- управление ресурсами.

В этой главе недостаточно места, чтобы рассмотреть все фреймворки ORM, поддерживаемые фреймворком Spring. Это даже хо-

рошо, потому что поддержка различных реализаций ORM в Spring похожа друг на друга. Как только вы приобретаете навык использования любого из фреймворков ORM в Spring, вы легко сможете переключиться на другой.

Начнем изучение с обзора интеграции Spring с самым, пожалуй, популярным фреймворком – Hibernate. Далее в этой главе (раздел 6.5) мы также познакомимся с особенностями интеграции Spring с JPA.

Hibernate – это открытый фреймворк ORM, получивший широкую популярность в сообществе разработчиков. Он обеспечивает не только базовые возможности объектно-реляционного отображения, но также и все прочие «фишки», которые принято ожидать от полнофункционального инструмента ORM, такие как отложенная загрузка, полная загрузка и распределенное кеширование.

В этом разделе мы сконцентрируемся на особенностях интеграции Spring с Hibernate, не углубляясь в запутанные детали использования фреймворка Hibernate. За более подробной информацией по работе с фреймворком Hibernate я рекомендую обращаться к книге «Java Persistence with Hibernate» (Manning, 2006) или на веб-сайт проекта Hibernate по адресу: <http://www.hibernate.org>.

6.4.1. Обзор *Hibernate*

В предыдущем разделе было показано, как использовать в приложении шаблоны JDBC, предоставляемые фреймворком Spring. Как оказывается, для работы с Hibernate фреймворк Spring предлагает похожий класс шаблона, абстрагирующий использование возможностей фреймворка Hibernate. Исторически для работы с фреймворком Hibernate в приложениях на основе Spring используется класс `HibernateTemplate`. Подобно своим сородичам из реализации поддержки JDBC, класс `HibernateTemplate` сам беспокоится обо всех тонкостях взаимодействий с Hibernate, перехватывая исключения, генерируемые этим фреймворком и преобразуя их в неконтролируемые исключения Spring.

Одной из областей ответственности класса `HibernateTemplate` является управление сессиями Hibernate. Сюда входят: открытие и закрытие сеанса, а также обеспечение уникальности сеанса для каждой транзакции. Без применения класса `HibernateTemplate` у вас не было бы иного выбора, как загромождать реализацию своих объектов DAO шаблонным кодом управления сессиями.



Недостатком класса `HibernateTemplate` является его некоторая навязчивость. При использовании класса `HibernateTemplate` в реализации объектов DAO (непосредственно или через класс поддержки `HibernateDaoSupport`) класс реализации DAO оказывается тесно привязанным к Spring API. Для кого-то это может оказаться не очень большой проблемой, но для других такая тесная связь с фреймворком Spring может оказаться нежелательной.

Даже при том, что класс `HibernateTemplate` все еще остается доступным, он больше не считается лучшим способом взаимодействия с Hibernate. В версии Hibernate 3 появились *контекстные сеансы* (contextual sessions), посредством которых Hibernate сам осуществляет управление сессиями `Session` и их распределением по одному на каждую транзакцию. Теперь нет никакой необходимости использовать класс `HibernateTemplate`, чтобы гарантировать это поведение, что избавляет классы DAO от необходимости писать программный код, накладывающий зависимость от Spring.

Поскольку контекстные сеансы признаны более удачным способом взаимодействия с фреймворком Hibernate, мы сконцентрируемся на них и не будем тратить время на знакомство с классом `HibernateTemplate`. Если же кому-то будет любопытно поближе познакомиться с `HibernateTemplate`, я рекомендую обратиться ко второму изданию этой книги или к примерам, которые можно загрузить по адресу: <http://www.manning.com/walls4/>, куда я включил примеры использования класса `HibernateTemplate`.

Прежде чем погрузиться в работу с контекстными сессиями Hibernate, необходимо познакомиться с особенностями настройки фабрики сеансов Hibernate в Spring.

6.4.2. Объявление фабрики сеансов Hibernate

Основным интерфейсом для взаимодействий с Hibernate является интерфейс `org.hibernate.Session`. Интерфейс `Session` обеспечивает базовую функциональность доступа к данным, позволяя сохранять, обновлять, удалять и загружать объекты в/из базы данных. Именно через интерфейс `Session` прикладные объекты DAO будут выполнять все операции с хранилищем данных.

Стандартный способ получить ссылку на объект `Session` – обратиться к реализации интерфейса `SessionFactory` в Hibernate. Среди

всего прочего интерфейс SessionFactory отвечает за открытие, закрытие и управление сеансами Hibernate.

Получить доступ к SessionFactory в приложениях на основе Spring можно через компоненты фабрики сеансов. Эти компоненты реализуют интерфейс FactoryBean фреймворка Spring, который воспроизводит объекты класса SessionFactory при внедрении в свойства типа SessionFactory. Это позволяет настраивать фабрику сеансов Hibernate наряду с другими компонентами в контексте приложения Spring.

Что касается настройки компонента фабрики сеансов Hibernate, имеются несколько вариантов на выбор. Решение зависит от того, желаете ли вы настроить сохранение объектов предметной области с использованием XML-файлов отображения или с помощью аннотаций. При выборе первого варианта на основе XML-файлов отображений необходимо настроить компонент LocalSessionFactoryBean в Spring:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
      <list>
        <value>Spitter.hbm.xml </value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="dialect">org.hibernate.dialect.HSQLDialect</prop>
      </props>
    </property>
  </bean>
```

Конфигурация компонента LocalSessionFactoryBean включает настройку трех свойств. В свойство dataSource внедряется ссылка на компонент DataSource. Свойство mappingResources принимает список из одного или более файлов отображений, определяющих стратегию хранения данных приложения. Наконец, свойство hibernateProperties позволяет определить мелкие детали поведения Hibernate. В данном случае фреймворку Hibernate сообщается, что он будет взаимодействовать с базой данных Hypersonic и для создания SQL-запросов должен использовать диалект HSQLDialect.



Если предпочтение будет отдано аннотациям, тогда вместо компонента LocalSessionFactoryBean следует использовать компонент AnnotationSessionFactoryBean:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.annotation.
      ↳AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="packagesToScan"
              value="com.habuma.spitter.domain" />
    <property name="hibernateProperties">
      <props>
        <prop key="dialect">org.hibernate.dialect.HSQLDialect</prop>
      </props>
    </property>
</bean>
```

Как и в случае с компонентом LocalSessionFactoryBean, свойства dataSource и hibernateProperties определяют место, где следует искать соединение с базой данных и ее тип.

Но вместо списка файлов отображений можно воспользоваться свойством packagesToScan, чтобы сообщить фреймворку Spring один или более пакетов, где следует искать классы объектов предметной области, аннотированные для сохранения с помощью Hibernate. В их число входят классы, отмеченные аннотациями JPA @Entity или @MappedSuperclass, и собственной аннотацией Hibernate – @Entity.

Список из одного элемента. Свойство packagesToScan компонента AnnotationSessionFactoryBean принимает массив строк, определяющих пакеты для поиска классов хранимых объектов. Обычно указывается список, как показано ниже:

```
<property name="packagesToScan">
  <list>
    <value>com.habuma.spitter.domain</value>
  </list>
</property>
```

Но когда поиск требуется выполнить в единственном пакете, можно воспользоваться преимуществом встроенного редактора свойства, который автоматически преобразует единственное строковое значение в массив строк.

При желании можно также явно перечислить все прикладные классы хранимых объектов, определив список полных имен классов в свойстве annotatedClasses:

```
<property name="annotatedClasses">
    <list>
        <value>com.habuma.spitter.domain.Spitter</value>
        <value>com.habuma.spitter.domain.Spittle</value>
    </list>
</property>
```

Свойство annotatedClasses отлично подходит для случаев, когда в приложении имеются два-три класса хранимых объектов. Но в других случаях, при большом количестве классов, когда было бы слишком утомительно определять полный их список или когда желательно оставить за собой свободу добавлять и удалять классы без изменения конфигурации Spring, удобнее использовать свойство packagesToScan.

Определив компонент фабрики сеансов Hibernate в контексте приложения Spring, можно приступать к созданию классов DAO.

6.4.3. Создание классов для работы с Hibernate, независимых от Spring

Как отмечалось выше, без контекстных сеансов гарантировать создание единственного сеанса для каждой транзакции можно с помощью шаблона для работы с Hibernate в Spring. Но теперь, когда управление сеансами берет на себя сам фреймворк Hibernate, нет необходимости использовать класс шаблона. Это означает возможность непосредственного связывания сеанса Hibernate с прикладными классами DAO.

Листинг 6.7. Контекстные сеансы позволяют определять классы DAO для работы с Hibernate, независимые от Spring

```
package com.habuma.spitter.persistence;
import java.util.List;
import org.hibernate.SessionFactory;
import org.hibernate.classic.Session;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;
```



```
import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;

@Repository
public class HibernateSpitterDao implements SpitterDao {
    private SessionFactory sessionFactory;

    @Autowired
    public HibernateSpitterDao(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;           // Конструирует DAO
    }

    private Session currentSession() {                // Извлекает текущий
        return sessionFactory.getCurrentSession();      // сеанс из фабрики
    }                                                 // SessionFactory

    public void addSpitter(Spitter spitter) {
        currentSession().save(spitter);               // Использует текущий сеанс
    }

    public Spitter getSpitterById(long id) {          // Использует текущий сеанс
        return (Spitter) currentSession().get(Spitter.class, id);
    }

    public void saveSpitter(Spitter spitter) {
        currentSession().update(spitter);              // Использует текущий сеанс
    }
    ...
}
```

Относительно фрагмента в листинге 6.7 следует сделать несколько замечаний. Во-первых, обратите внимание на использование аннотации Spring `@Autowired`, вынуждающей Spring автоматически внедрить `SessionFactory` в свойство `sessionFactory` объекта `HibernateSpitterDao`. Этот объект `SessionFactory` используется затем в методе `currentSession()`, чтобы получить ссылку на сеанс для текущей транзакции.

Отметьте также, что класс отмечен аннотацией `@Repository`. Это позволяет достичь двух целей. Во-первых, `@Repository` – еще одна стереотипная аннотация в Spring, которая, кроме всего прочего, обнаруживается элементом конфигурации Spring `<context:component-scan>`. Это означает, что при наличии настроенного элемента `<context:component-scan>`, как показано ниже, не требуется явно объявлять компонент `HibernateSpitterDao`:

```
<context:component-scan  
    base-package="com.habuma.spitter.persistence" />
```

Вторая цель, преследуемая аннотацией `@Repository`, – уменьшить объем кода разметки XML в файле конфигурации. Напомню, что одной из задач класса шаблона являются перехват специфических исключений и их преобразование в неконтролируемые, универсальные исключения Spring. Но как обеспечить такое преобразование после отказа от класса шаблона и перехода к использованию контекстных сеансов Hibernate?

Чтобы добавить преобразование исключений без применения класса шаблона поддержки Hibernate, достаточно просто добавить в контекст приложения Spring компонент `PersistenceExceptionTranslationPostProcessor`:

```
<bean class="org.springframework.dao.annotation.  
    ↪PersistenceExceptionTranslationPostProcessor"/>
```

`PersistenceExceptionTranslationPostProcessor` – это механизм постобработки компонентов, добавляющий объект-советник во все компоненты, отмеченные аннотацией `@Repository`, который будет перехватывать все специфические исключения и преобразовывать их в соответствующие неконтролируемые исключения Spring.

На этом реализация Hibernate-версии нашего класса DAO заканчивается. Нам удалось определить класс, избежав прямой зависимости от каких-либо классов Spring (за исключением аннотации `@Repository`). Аналогичный подход без применения классов шаблонов можно применить при разработке классов DAO на основе JPA. Поэтому сделаем еще одно усилие и создадим еще одну реализацию интерфейса `SpitterDao`, но на этот раз с использованием JPA.

6.5. Spring и Java Persistence API

С самого начала спецификация EJB включала понятие компонентов-сущностей (entity beans). В терминах EJB *компонент-сущность* представляет собой тип EJB, описывающий прикладные объекты, хранимые в реляционной базе данных. Компоненты-сущности претерпели несколько этапов развития на протяжении последних лет, включая появление компонентов-сущностей, которые *сами управляют своим сохранением* (bean-managed persistence, BMP), и ком-



понентов-сущностей, сохранением которых управляет контейнер (container-managed persistence, CMP).

Компоненты-сущности пережили взлет и падение популярности EJB. В последние годы разработчики все чаще отказываются от тяжеловесной платформы EJB, отдавая предпочтение реализациям на основе простых POJO. Это явилось серьезной проблемой для организации Java Community Process, формировавшей новую спецификацию EJB на основе POJO. В итоге появилась спецификация JSR-220, также известная как *EJB 3*.

На руинах спецификации компонентов-сущностей EJB 2 сформировался новый стандарт хранения данных в Java – Java Persistence API (JPA). JPA представляет собой механизм хранения данных на основе POJO, заимствующий идеи Hibernate и *Java Data Objects* (JDO) и дополняющий их аннотациями Java 5.

С выходом версии Spring 2.0 состоялась премьера интеграции Spring с JPA. По иронии, многие обвиняют (или приписывают) Spring в упадке EJB. Но теперь, после появления поддержки JPA, многие разработчики рекомендуют использовать в приложениях на основе Spring для работы с базами данных именно JPA. В действительности многие считают, что связка Spring-JPA – это мечта для разработки на основе POJO.

Первый шаг к использованию JPA в Spring заключается в настройке компонента фабрики диспетчера сущностей в контексте приложения.

6.5.1. Настройка фабрики диспетчера сущностей

Если говорить в двух словах, приложения на основе JPA используют реализацию EntityManagerFactory для получения экземпляра EntityManager. Спецификация JPA определяет два вида диспетчеров сущностей (entity managers).

- Управляемые приложением – эти диспетчеры сущностей создаются, когда приложение непосредственно запрашивает у фабрики диспетчеров сущностей. За создание и уничтожение диспетчеров сущностей, управляемых приложением, а также за их использование в транзакциях отвечает само приложение. Этот тип диспетчеров сущностей в большей степени подходит для использования в автономных приложениях, выполняющихся вне контейнера Java EE.

- ❑ Управляемые контейнером – эти диспетчеры сущностей создаются и управляются контейнером Java EE. Приложение никак не взаимодействует с фабрикой диспетчеров сущностей. Вместо этого диспетчеры сущностей приобретаются приложением посредством внедрения или из JNDI. За настройку фабрик диспетчеров сущностей отвечает контейнер. Этот тип диспетчеров сущностей в большей степени подходит для использования контейнером Java EE, когда требуется обеспечить некоторый контроль над настройками JPA, помимо тех, что определены в файле persistence.xml.

Оба типа диспетчеров сущностей реализуют один и тот же интерфейс EntityManager. Но основное отличие заключается не в EntityManager как таковом, а скорее в том, как диспетчер создается и управляется. Диспетчеры сущностей, управляемые приложением, создаются объектом EntityManagerFactory, полученным вызовом метода createEntityManagerFactory() объекта PersistenceProvider, а диспетчеры сущностей, управляемые контейнером, – вызовом метода createContainerEntityManagerFactory().

Так что же все это значит для разработчиков приложений на основе Spring, желающих использовать JPA? В сущности, не так много. Независимо от способа создания EntityManagerFactory, ответственность за управление диспетчерами сущностей будет нести Spring. При использовании диспетчеров сущностей, управляемых приложением, Spring будет играть роль приложения и обеспечит прозрачное управление диспетчерами от имени приложения. В случае использования диспетчеров сущностей, управляемых контейнером, Spring будет играть роль контейнера.

Каждая фабрика диспетчеров сущностей создается соответствующим компонентом Spring:

- ❑ LocalEntityManagerFactoryBean создает фабрики EntityManagerFactory, управляемые приложением;
- ❑ LocalContainerEntityManagerFactoryBean создает фабрики EntityManagerFactory, управляемые контейнером.

Важно отметить, что выбор между фабриками, управляемыми приложением и контейнером, полностью прозрачен для приложений на основе Spring. Класс JpaTemplate фреймворка Spring полностью скрывает все сложности использования любой из форм EntityManagerFactory, позволяя сосредоточиться в прикладном коде на его основной цели – реализации доступа к данным.

Единственное существенное отличие между ними состоит в том, как они настраиваются в контексте приложения Spring. Рассмотрим для начала, как настроить в Spring компонент LocalEntityManagerFactoryBean фабрики диспетчеров сущностей, управляемых приложением. А затем проделаем то же с фабрикой диспетчеров сущностей, управляемых контейнером LocalContainerEntityManagerFactoryBean.

Настройка механизма JPA, управляемого приложением

Большая часть настроек фабрики диспетчеров сущностей, управляемых приложением, находится в файле с именем persistence.xml. Этот файл должен находиться в каталоге META-INF, в библиотеке классов.

Назначение файла persistence.xml – определить одну или более единиц хранения. Каждая единица хранения представляет собой группу из одного или более классов хранимых объектов, соответствующих одному источнику данных. То есть в файле persistence.xml перечисляются классы хранимых объектов наряду с дополнительными настройками, такими как настройки источников данных и ссылки на XML-файлы отображений. Ниже приводится пример типичного файла persistence.xml, принадлежащего приложению Spitter:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    version="1.0">
    <persistence-unit name="spitterPU">
        <class>com.habuma.spitter.domain.Spitter</class>
        <class>com.habuma.spitter.domain.Spittle</class>
        <properties>
            <property name="toplink.jdbc.driver"
                value="org.hsqldb.jdbcDriver" />
            <property name="toplink.jdbc.url"
                value="jdbc:hsqldb:hsqldb://localhost/spitter/spitter" />
            <property name="toplink.jdbc.user"
                value="sa" />
            <property name="toplink.jdbc.password"
                value="" />
        </properties>
    </persistence-unit>
</persistence>
```

Из-за того, что основные настройки сосредоточены в файле persistence.xml, в файле конфигурации Spring требуется (если вообще требуется) указать совсем немного настроек. Следующий элемент <bean> объявляет компонент LocalEntityManagerFactoryBean в Spring:

```
<bean id="emf"
      class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="spitterPU" />
</bean>
```

Значение, указанное в свойстве `persistenceUnitName`, ссылается на имя единицы хранения, объявленной в файле `persistence.xml`.

Причина, почему большая часть настроек создания фабрик `EntityManagerFactory` диспетчеров сущностей, управляемых приложением, сосредоточена в файле `persistence.xml`, обусловлена стремлением переместить в него все, что связано с созданием фабрик. При использовании сущностей, управляемых приложением (без привлечения фреймворка Spring), полную ответственность за получение фабрики `EntityManagerFactory` через реализацию `PersistenceProvider` несет само приложение. Код приложения окажется невероятно раздутым, если будет определять единицы хранения при каждой попытке получить объект `EntityManagerFactory`. При наличии же настроек в файле `persistence.xml` механизм JPA сможет без труда отыскать определения единиц хранения в известном месте.

Однако благодаря поддержке JPA в Spring вам никогда не придется напрямую работать с `PersistenceProvider`. Поэтому кажется бессмысленным перемещение конфигурационной информации в файл `persistence.xml`. Фактически это препятствует определению настроек `EntityManagerFactory` в конфигурации Spring (чтобы, например, определить источник данных, настраиваемый фреймворком Spring).

Поэтому имеет смысл обратить внимание на механизм JPA, управляемый контейнером.

Настройка механизма JPA, управляемого контейнером

Настройка механизма JPA, управляемого контейнером, предполагает несколько иной подход. При выполнении в пределах контейнера `EntityManagerFactory` может быть создан на основе информации, предоставляемой контейнером, в данном случае – контейнером Spring.

Вместо настройки источника данных в файле `persistence.xml` необходимую информацию можно определить в контексте приложения Spring. Например, следующее объявление элемента `<bean>` демонстрирует, как настроить в Spring механизм JPA, управляемый контейнером, используя `LocalContainerEntityManagerFactoryBean`.

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

Здесь в свойство `dataSource` внедряется ссылка на источник данных, настраиваемый в конфигурации Spring. В данном случае можно использовать любую реализацию `javax.sql.DataSource`, например реализацию, настроенную в разделе 6.2. Источник данных все еще может настраиваться в `persistence.xml`, однако источник данных, определяемый через это свойство, имеет более высокий приоритет.

Свойство `jpaVendorAdapter` может быть использовано для настройки особенностей конкретной реализации JPA. В состав Spring входят несколько классов адаптеров JPA:

- EclipseLinkJpaVendorAdapter;
- HibernateJpaVendorAdapter;
- OpenJpaVendorAdapter;
- TopLinkJpaVendorAdapter.

В данном случае использована JPA-реализация для Hibernate, поэтому настройки выполняются с применением `HibernateJpaVendorAdapter`:

```
<bean id="jpaVendorAdapter"
    class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
    <property name="database" value="HSQL" />
    <property name="showSql" value="true"/>
    <property name="generateDdl" value="false"/>
    <property name="databasePlatform"
        value="org.hibernate.dialect.HSQLDialect" />
</bean>
```

Таблица 6.5. Адаптер *Hibernate JPA* поддерживает несколько типов баз данных. Определить тип базы данных можно настройкой его свойства

База данных	Значение для свойства <code>database</code>
IBM DB2	DB2
Apache Derby	DERBY
H2	H2
Hypersonic	HSQL

Таблица 6.5 (окончание)

База данных	Значение для свойства database
Informix	INFORMIX
MySQL	MYSQL
Oracle	ORACLE
PostgreSQL	POSTGRESQL
Microsoft SQL Server	SQLSERVER
Sybase	SYBASE

Адаптер имеет несколько свойств, доступных для настройки, но наиболее важным из них является свойство `database`, где в данном примере определен тип используемой базы данных – Hypersonic. Другие значения, поддерживаемые этим свойством, перечислены в табл. 6.5.

Некоторые динамические особенности хранилищ данных требуют, чтобы классы хранимых объектов оснащались поддержкой этих особенностей. Классы объектов, свойства которых загружаются по требованию (то есть не загружаются из базы данных до явного обращения к ним), должны поддерживать извлечение незагруженных данных при обращении к ним. Для поддержки загрузки по требованию некоторые фреймворки используют динамические прокси-объекты. Другие, такие как JDO, оснащают классы необходимой поддержкой во время компиляции.

Выбор того или иного компонента фабрики диспетчеров существенно зависит в первую очередь от того, как он будет использоваться. Для простых приложений достаточно будет использовать `LocalEntityManagerFactoryBean`. Но, поскольку `LocalContainerEntityManagerFactoryBean` обеспечивает более широкие возможности настройки механизма JPA в Spring, он выглядит более привлекательно, и вы наверняка предпочтете использовать именно его в своих приложениях.

Получение EntityManagerFactory из JNDI

Следует также отметить, что при развертывании приложения на основе Spring на некоторых серверах приложений компонент `EntityManagerFactory` уже может быть создан автоматически и находится в JNDI, ожидая, пока его извлекут. В этом случае для получения ссылки на `EntityManagerFactory` можно использовать элемент `<jee:jndi-lookup>` из пространства имен `jee`:

```
<jee:jndi-lookup id="emf" jndi-name="persistence/spitterPU" />
```



Независимо от того, как будет получен компонент EntityManagerFactory, после его приобретения можно приступать к созданию классов DAO. Сделаем это прямо сейчас.

6.5.2. Объект DAO на основе JPA

Подобно всем остальным модулям Spring интеграции с механизмами хранения данных, модуль Spring JPA реализован в форме класса шаблона JpaTemplate и соответствующего ему класса поддержки JpaDaoSupport. Однако мы не будем рассматривать приемы использования JPA на основе шаблона и отдадим предпочтение подходу, основанному на применении JPA без тесной связи с фреймворком Spring. Этот подход можно сравнить с контекстными сессиями Hibernate, которые использовались в разделе 6.4.3.

Поскольку нас больше интересует подход к использованию JPA без тесной связи с фреймворком Spring, в этом разделе мы сфокусируемся на создании объектов JPA DAO, также не связанных с фреймворком Spring. В частности, в листинге 6.8 представлен класс JpaSpitterDao, не использующий шаблона JpaTemplate.

Листинг 6.8. Объект JPA DAO, не использующий шаблонов Spring

```
package com.habuma.spitter.persistence;
import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

import org.springframework.dao.DataAccessException;
import org.springframework.stereotype.Repository;
import org.springframework.transaction.annotation.Transactional;

import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;

@Repository("spitterDao")
@Transactional
public class JpaSpitterDao implements SpitterDao {
    private static final String RECENT_SPITTLES =
        "SELECT s FROM Spittle s";
    private static final String ALL_SPITTERS =
        "SELECT s FROM Spitter s";
    private static final String SPITTER_FOR_USERNAME =
        "SELECT s FROM Spitter s WHERE s.username = ?";
```

```
"SELECT s FROM Spitter s WHERE s.username = :username";
private static final String SPITTLES_BY_USERNAME =
    "SELECT s FROM Spittle s WHERE s.spitter.username = :username";

@PersistenceContext
private EntityManager em; // Для внедрения EntityManager

public void addSpitter(Spitter spitter) {
    em.persist(spitter); // Использование EntityManager
}

public Spitter getSpitterById(long id) {
    return em.find(Spitter.class, id); // Использование EntityManager
}

public void saveSpitter(Spitter spitter) {
    em.merge(spitter); // Использование EntityManager
}
...
}
```

Для взаимодействия с хранилищем класс JpaSpitterDao использует EntityManager. Благодаря использованию EntityManager объект DAO остается независимым от фреймворка Spring и напоминает объекты DAO, которые можно встретить в приложениях, не использующих фреймворка Spring. Но как он получает экземпляр EntityManager?

Обратите внимание, что свойство em отмечено аннотацией @PersistenceContext. Эта аннотация явно указывает, что в свойство em должен быть внедрен экземпляр EntityManager. Чтобы обеспечить внедрение EntityManager средствами Spring, необходимо сконфигурировать компонент PersistenceAnnotationBeanPostProcessor в контексте приложения Spring:

```
<bean class="org.springframework.orm.jpa.support.
    ↳PersistenceAnnotationBeanPostProcessor"/>
```

Возможно, вы также заметили, что класс JpaSpitterDao отмечен аннотациями @Repository и @Transactional. Аннотация @Transactional указывает, что методы взаимодействий с хранилищем в этом объекте DAO будут выполняться в контексте транзакций. Подробнее об аннотации @Transactional будет рассказываться в следующей главе, где будет обсуждаться поддержка декларативных транзакций в Spring.



Что касается аннотации `@Repository`, здесь она служит той же цели, что и в примере реализации DAO с использованием контекстных сеансов Hibernate. В отсутствие шаблона, выполняющего преобразование исключений, необходимо использовать аннотацию `@Repository`, чтобы компонент `PersistenceExceptionTranslationPostProcessor` знал, что это один из компонентов, требующих преобразования специализированных исключений в универсальные исключения Spring.

Говоря о `PersistenceExceptionTranslationPostProcessor`, следует помнить, что его необходимо настроить как компонент Spring, как это делалось в примере с Hibernate:

```
<bean class="org.springframework.dao.annotation.  
    ↳PersistenceExceptionTranslationPostProcessor"/>
```

Обратите внимание, что преобразование исключений при использовании JPA или Hibernate не является обязательным требованием. Если вы предпочитаете получать исключения, специфические для JPA или Hibernate, тогда просто опустите настройку компонента `PersistenceExceptionTranslationPostProcessor` и используйте специализированные исключения. Но помните, что использование механизма преобразования исключений в Spring обеспечивает унификацию всех исключений доступа к данным, что упростит замену одних механизмов хранения данных другими.

6.6. Кеширование

Во многих приложениях данные читаются значительно чаще, чем записываются. В нашем приложении RoadRantz, например, большинство посетителей сайта заинтересованы в получении той или иной информации, чем в написании собственных сообщений. Хотя, безусловно, список сообщений со временем будет расти, но просматриваться он будет все же значительно чаще.

Кроме того, данные, представляемые приложением RoadRantz, не так чувствительны ко времени. Если посетитель увидит немного устаревший список сообщений, у него не сложится отрицательного впечатления. В конечном счете они смогут вернуться на сайт позднее и увидеть обновленный список сообщений.

Всякий раз, когда запрашивается список сообщений, объект DAO обращается к базе данных и запрашивает последние данные (чаще всего те же, что были на момент последнего обращения).

Операции с базами данных нередко являются наиболее узким местом в приложениях с точки зрения их производительности. Даже самые простые запросы к самым оптимизированным хранилищам данных могут добавить проблем производительности в высоконагруженном приложении.

Когда данные в приложении часто запрашиваются, но редко обновляются, кажется странным все время запрашивать из базы данных самую последнюю информацию. Куда предпочтительнее выглядит применение механизма кеширования данных.

На первый взгляд решение проблемы кеширования данных выглядит достаточно просто: после получения некоторой информации сохранить ее, чтобы в дальнейшем было возможно обратиться к ней. Однако реализация кеша может показаться довольно трудоемкой. Например, взгляните на реализацию метода `getRantsForDay()` класса `HibernateRantDao`.

```
public List<Rant> getRantsForDay(Date day) {
    return getHibernateTemplate().find("from " + RANT +
        " where postedDate = ?", day);
}
```

Метод `getRantsForDay()` является идеальным кандидатом для реализации кеширования в нем, поскольку нет способа вернуться в прошлое и добавить сообщение задним числом. Если запрашивается список на текущий день, то он неизменен. Таким образом, не имеет никакого смысла каждый раз обращаться к базе данных за списком сообщений, которые были размещены, скажем, в прошлый вторник. Запрос к базе данных должен выполняться только раз, и тогда мы сможем запомнить данные, чтобы вернуть их, когда они потребуются снова.

Теперь изменим метод `getRantsForDay()`, добавив в него поддержку доморощенного кеша:

```
public List<Rant> getRantsForDay(Date day) {
    List<Rant> cachedResult =
        rantCache.lookup("getRantsForDay", day);
    if(cachedResult != null) {
        return cachedResult;
    }

    cachedResult = getHibernateTemplate().find("from " + RANT +
```

```

    " where postedDate = ? ", day);

rantCache.store("getRantsForDay", day, cachedResult);

return cachedResult
}

```

Эта версия метода `getRantsForDay()` выглядит гораздо более неподдельной. Главная цель метода `getRantsForDay()` – в том, чтобы найти сообщения, относящиеся к указанному дню, но большая часть метода реализует кеширование. Кроме того, наша реализация не предусматривает решения некоторых проблем, связанных с кешированием, таких как истечение срока хранения информации в кеше,бросок кеша и обработка его переполнения.

К счастью, в Spring доступно гораздо более элегантное решение кеширования. Проект Spring Modules (<http://springmodules.dev.java.net>) обеспечивает поддержку кеширования через аспекты. Вместо того чтобы предусматривать явную реализацию кеширования в методах, аспекты кеширования Spring Modules предоставляют возможность применять советы к методам компонентов, обеспечивающие прозрачное кеширование результатов.

Как показано на рис. 6.5, поддержка кеширования в Spring включает в себя прокси-объект, который перехватывает вызовы к одному или нескольким методам компонентов, управляемых фреймворком Spring. При вызове метода прокси-объекта модуль Spring Modules Cache сначала проверяет, вызывался ли этот метод прежде с теми же аргументами. Если вызывался, то фактический метод не вызывается



Рис. 6.5. Модуль кеширования Spring Modules
перехватывает вызовы методов компонента и отыскивает данные,
чем обеспечивает более высокую скорость доступа к данным
в сравнении с медленными запросами к базе данных

и возвращается содержимое кеша. В противном случае вызывается фактический метод, а его возвращаемое значение сохраняется в кеше, чтобы использовать его при следующем обращении к методу.

В этом разделе мы добавим поддержку кеширования в слой доступа к данным для приложения RoadRantz с помощью модуля Spring Modules Cache. Это обеспечит нашему приложению более высокую производительность и даст заслуженный отдых базе данных.

6.6.1. Настройка кеширования

Несмотря на то что модуль Spring Modules Cache предоставляет прокси-объект для перехвата вызовов методов, сохраняющий результаты в кеше, он не имеет собственной реализации кеша. Вместо этого опирается на использование сторонних решений. Модулем поддерживаются несколько механизмов кеширования, включая следующие:

- EHCache;
- GigaSpaces;
- JBoss Cache;
- JCS;
- OpenSymphony OSCache;
- Tangosol Coherence.

Для использования в примере приложения RoadRantz я выбрал механизм EHCache. Это решение основано в первую очередь на моем собственном опыте работы с EHCache и на том обстоятельстве, что это решение свободно доступно в репозитории Maven по адресу: www.ibiblio.org. Однако независимо от выбора настройки модуля Spring Modules Cache будут очень похожи для любых решений.

В первую очередь необходимо создать новый конфигурационный файл Spring, куда будут помещаться настройки механизма кеширования. В принципе, настройки Spring Modules Cache можно поместить в любой конфигурационный файл Spring, загружаемый вместе с приложением RoadRantz, но их лучше хранить отдельно. Поэтому создадим файл `roadrantz-cache.xml`, куда поместим параметры настройки механизма кеширования.

Как и в любых других конфигурационных файлах Spring, корневым элементом в файле `roadrantz-cache.xml` должен быть элемент `<beans>`. Однако, чтобы воспользоваться преимуществами поддержки EHCache в Spring Modules, в элемент `<beans>` следует добавить объявление пространства имен `ehcache`:



```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:ehcache="http://www.springmodules.org/schema/ehcache"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://www.springmodules.org/schema/ehcache
                           http://www.springmodules.org/schema/cache/
                           ↳ springmodules-ehcache.xsd">
...
</beans>
```

В приложении RoadRantz предполагается использовать механизм кеширования EHCache, но если вы отдастите предпочтение другому механизму, следует заменить объявления пространства имен и схемы на соответствующие вашему выбору. В табл. 6.6 перечислены все пространства имен, а также URI пространств имен и соответствующих им схем.

Независимо от выбранного механизма кеширования для его настройки будут использоваться одни и те же элементы конфигурирования Spring, которые перечислены в табл. 6.7.

Таблица 6.6. Пространства имен и схемы различных механизмов кеширования, поддерживаемых модулем Spring Modules

Пространство имен	URI пространства имен	URI схемы
ehcache	http://www.springmodules.org/schema/ehcache	http://www.springmodules.org/schema/cache/springmodules-ehcache.xsd
gigaspaces	http://www.springmodules.org/schema/gigaspaces	http://www.springmodules.org/schema/cache/springmodulesgigaspaces.xsd
jboss	http://www.springmodules.org/schema/jboss	http://www.springmodules.org/schema/cache/springmodulesjboss.xsd
jcs	http://www.springmodules.org/schema/jcs	http://www.springmodules.org/schema/cache/springmodules-jcs.xsd
oscache	http://www.springmodules.org/schema/oscache	http://www.springmodules.org/schema/cache/springmodulesoscache.xsd
tangosol	http://www.springmodules.org/schema/tangosol	http://www.springmodules.org/schema/cache/springmodulestangosol.xsd

Таблица 6.7. Элементы конфигурирования Spring Modules

Конфигурационный элемент	Назначение
<namespace:annotations>	Объявляет кешированные методы, помечая их аннотациями Java 5
<namespace:commons-attributes>	Объявляет кешированные методы, помечая их метаданными Jakarta Commons Attributes
<namespace:config>	Содержит настройки механизма кеширования EHCache в файлах конфигурации Spring
<namespace:proxy>	Объявляет кешированные методы, определяя прокси-объект в XML-конфигурации Spring

Поскольку в примере решено использовать механизм кеширования EHCache, необходимо сообщить фреймворку Spring, где искать конфигурационный файл EHCache¹. Воспользуемся для этого элементом <ehcache:config>:

```
<ehcache:config
    configLocation="classpath:ehcache.xml" />
```

Атрибут configLocation в этом фрагменте сообщает фреймворку Spring, что файл с настройками EHCache следует загружать из корня библиотеки классов (classpath) приложения.

Настройка EHCache

Внешний файл ehcache.xml для нашего примера содержит настройки, как показано в листинге 6.9.

Листинг 6.9. Настройки механизма EHCache в файле ehcache.xml

```
<ehcache>
    <defaultCache
        maxElementsInMemory="500"
        overflowToDisk="false"
        memoryStoreEvictionPolicy="LFU"
        eternal="true" />           <!-- Настройка кеша по умолчанию -->
```

¹ На момент написания этих строк конфигурацию EHCache (и других механизмов кэширования) все еще приходилось определять во внешних XML-файлах, используя наборы элементов, характерные для каждого механизма. Но в будущих версиях, возможно, появится поддержка определения настроек с помощью элемента <namespace:config>, что избавит от необходимости использовать внешние файлы.



```
<cache name="rantzCache"
      maxElementsInMemory="500"
      overflowToDisk="false"
      memoryStoreEvictionPolicy="LFU" />
      eternal="true"           <!-- Настройки кеша rantzCache --&gt;
&lt;/ehcache&gt;</pre>


---



```

В этом фрагменте выполнена настройка двух кешей, управляемых механизмом EHCache. Элемент `<defaultCache>` является обязательным и описывает кеш, который будет использоваться, когда не будет найден более подходящий кеш. Элемент `<cache>` определяет другие кеши и может содержаться в файле `ehcache.xml` ноль или более раз (по одному для каждого кеша). Здесь определяется только один кеш, используемый не по умолчанию, – `rantzCache`.

Атрибуты элементов `<defaultCache>` и `<cache>` описывают поведение кешей. В табл. 6.8 перечислены атрибуты, которые можно использовать при настройке кешей в EHCache.

Таблица 6.8. Атрибуты настройки кеша в EHCache

Атрибут	Назначение
<code>diskExpiryThreadIntervalSeconds</code>	Интервал (в секундах) удаления устаревших записей из дискового кеша. (Значение по умолчанию: 120 сек.)
<code>diskPersistent</code>	Восстанавливать ли данные из дискового кеша после перезапуска VM. (Значение по умолчанию: <code>false</code>)
<code>eternal</code>	Следует ли рассматривать элементы в кеше как «вечные». Срок хранения «вечных» элементов никогда не истечет. (Обязательный)
<code>maxElementsInMemory</code>	Максимальное число элементов кеша, хранимых в памяти. (Обязательный)
<code>memoryStoreEvictionPolicy</code>	Алгоритм вытеснения элементов кеша из памяти по достижении предела <code>maxElementsInMemory</code> . По умолчанию применяется алгоритм с вытеснением по наибольшему времени неиспользования (<code>LRU</code>). Доступны также алгоритм «первым пришел, первым ушел» (<code>FIFO</code>) и алгоритм с вытеснением по меньшей частоте использования (<code>LFU</code>). (Значение по умолчанию: <code>LRU</code>)
<code>name</code>	Имя кеша. (Обязательный)
<code>overflowToDisk</code>	Допускается ли сбрасывать элементы в дисковый кеш из памяти по достижении предела <code>maxElementsInMemory</code> . (Обязательный)

Таблица 6.8 (окончание)

Атрибут	Назначение
timeToIdleSeconds	Интервал (в секундах) между обращениями к элементу кеша, прежде чем истечет срок его хранения. Значение 0 указывает, что элементы будут храниться в кеше независимо от величины интервала между обращениями. (Значение по умолчанию: 0)
timeToLiveSeconds	Время (в секундах), в течение которого элемент может оставаться в кеше, прежде чем истечет срок его хранения. Значение 0 указывает, что элементы будут храниться в кеше «вечно». (Значение по умолчанию: 0)

Для приложения RoadRantz был настроен один кеш по умолчанию (потому что EHCache требует его наличия) и еще один кеш с именем `rantzCache`, который будет играть роль основного кеша. Оба кеша настроены так, что они будут хранить до 500 элементов («вечно») с вытеснением по наибольшему времени неиспользования. Кроме того, конфигурация не предусматривает возможного сохранения вытесняемых элементов на диске¹.

Закончив настройку механизма EHCache в контексте приложения Spring, можно перейти к объявлению компонентов и их методов, результаты выполнения которых должны кешироваться. Начнем с объявления прокси-объекта, который будет выполнять кеширование значений, возвращаемых методами объектов DAO в приложении RoadRantz.

6.6.2. Настройка компонентов для кеширования

Выше мы уже определили, что метод `getRantsForDay()` класса `HibernateRantDao` является первым кандидатом на кеширование. Вернемся к определению контекста приложения Spring и обернем элементом `<ehcache:proxy>` объект класса `HibernateRantDao`, чтобы обеспечить кеширование всех результатов, возвращаемых его методом `getRantsForDay()`:

¹ Эти настройки были выбраны довольно произвольно, но они достаточно разумны. Естественно, при подготовке приложения к развертыванию необходимо рассмотреть разные варианты его использования и определить наиболее подходящие настройки.

```
<ehcache:proxy id="rantDao">
    refId="rantDaoTarget">
<ehcache:caching>
    methodName="getRantsForDay"
    cacheName="rantzCache" />
</ehcache:proxy>
```

Элемент `<ehcache:caching>` определяет методы, вызовы которых должны перехватываться и для которых должны возвращаться кешированные значения. В данном случае атрибут `methodName` определяет имя `getRantsForDay()` кешируемого метода и используемый для этих целей кеш `rantzCache`.

В элемент `<ehcache:proxy>` можно поместить сколько угодно элементов `<ehcache:caching>`, описывающих методы компонента. Можно добавить по одному элементу `<ehcache:caching>` на каждый кешируемый метод или использовать шаблонные символы в именах, чтобы единственным элементом `<ehcache:caching>` определить целую группу методов. Следующий элемент `<ehcache:caching>` включает кеширование для всех методов с именами, начинающимися со слова `get`:

```
<ehcache:caching>
    methodName="get*"
    cacheName="rantzCache" />
```

Добавление результатов в кеш – это лишь половина проблемы. Спустя некоторое время кеш заполнится большим количеством данных, часть которых может оказаться просто ненужной. В конечном итоге может появиться потребность очистить кеш и начать заполнять его заново. Посмотрим, как можно выполнить сброс кеша в результате вызова метода.

Сброс кеша

В то время как элемент `<ehcache:caching>` объявляет методы, результаты которых сохраняются в кеше, элемент `<ehcache:flushing>` объявляет методы, вызовы которых очищают кеш. Например, предположим, что необходимо организовать сброс кеша `rantzCache` при вызове метода `saveRant()`. Необходимые для этого настройки содержит следующий элемент `<ehcache:flushing>`:

```
<ehcache:flushing>
    methodName="saveRant"
    cacheName="rantzCache" />
```

По умолчанию кеш, определяемый атрибутом `cacheName`, будет сброшен после вызова метода, имя которого указано в атрибуте `methodName`, но есть возможность изменить момент сброса кеша, определив его в атрибуте `when`:

```
<ehcache:flushing
    methodName="saveRant"
    cacheName="rantzCache"
    when="before" />
```

Значение `before` в атрибуте `when` требует, чтобы кеш сбрасывался перед вызовом метода `saveRant()`.

Настройка кеширования для вложенных компонентов

Обратите внимание на атрибуты `id` и `refId` элемента `<ehcache:proxy>`. Прокси-объекту, создаваемому элементом `<ehcache:proxy>`, будет присвоен идентификатор `rantDao`. Однако это идентификатор существующего компонента типа `HibernateRantDao`. Поэтому необходимо переименовать существующий компонент в `rantDaoTarget` с помощью атрибута `refId`. (Это не противоречит особенностям именования прокси- и целевых объектов в классическом аспектно-ориентированном программировании в Spring.)

Если применение атрибутов `id/refId` кажется вам неуклюжим, тогда можно объявить целевой компонент вложенным по отношению к элементу `<ehcache:proxy>`. Например, следующий элемент `<ehcache:proxy>` определяет компонент типа `HibernateRantDao` как вложенный компонент:

```
<ehcache:proxy id="rantDao">

    <bean class="com.roadrantz.dao.HibernateRantDao">
        <property name="sessionFactory"
                  ref="sessionFactory" />
    </bean>

    <ehcache:caching
        methodName="getRantsForDay"
        cacheName="rantzCache" />
</ehcache:proxy>
```

Использование вложенных компонентов не избавляет от необходимости объявлять по одному элементу `<ehcache:proxy>` для каждого



кешируемого компонента и по одному или более `<ehcache:caching>` элементов для методов. Для простых приложений это совсем несложно. Но с ростом кешируемых компонентов и методов XML-файл конфигурации Spring будет разбухать все больше и больше.

Если даже подход на основе вложенных компонентов покажется неудобным или потребуется обеспечить кеширование нескольких компонентов, можно подумать об использовании поддержки декларативного кеширования с помощью аннотаций Spring Modules. Скажем элементу `<ehcache:proxy>` «прощай» и посмотрим, как Spring Modules обеспечивает поддержку декларативного кеширования с помощью аннотаций.

6.6.3. Декларативное кеширование с помощью аннотаций

Помимо возможности настройки кеширования в конфигурационном XML-файле, описанной в предыдущем разделе, модуль Spring Modules поддерживает декларативное кеширование с использованием метаданных на уровне программного кода. Эта поддержка доступна в двух вариантах:

- ❑ аннотации *Java 5* – идеальное решение, если целевой платформой для приложения является Java 5;
- ❑ *Jakarta Commons Attributes* – если целевой платформой являются версии Java, предшествующие версии Java 5.

Целевой платформой для приложения RoadRantz является Java 5. Поэтому для настройки кеширования в слое доступа данных будут использоваться аннотации Java 5. Модуль Spring Modules предусматривает две аннотации, имеющие отношение к кешированию:

- ❑ `@Cacheable` – отмечает метод, возвращаемое значение которого должно кешироваться;
- ❑ `@CacheFlush` – отмечает метод, вызов которого приводит к очистке кеша.

С помощью аннотации `@Cacheable` можно обеспечить кеширование метода `getRantsForDay()`, как показано ниже:

```
@Cacheable(modelId="rantzCacheModel")
public List<Rant> getRantsForDay(Date day) {
    return getHibernateTemplate().find("from " + RANT +
        " where postedDate = ?", day);
}
```

Атрибут `modelId` определяет модель кеширования значений, возвращаемых методом `getRantsForDay()`. Подробнее о том, как определяется модель кеширования, будет рассказано чуть ниже. Но сначала, с помощью аннотации `@CacheFlush`, определим операцию очистки кеша, которая будет выполняться при вызове метода `saveRant()`:

```
@CacheFlush(modelId="rantzFlushModel")
public void saveRant(Rant rant) {
    getHibernateTemplate().saveOrUpdate(rant);
}
```

Атрибут `modelId` определяет модель очистки кеша, которая будет выполняться при вызове метода `saveRant()`.

Что касается моделей кеширования и очистки кеша, вы, вероятно, хотели бы знать, как они определяются. Элемент `<ehcache:annotations>` используется для включения поддержки аннотаций кеширования в модуле Spring Modules. В файле `roaddrantzcache.xml` он объявлен, как показано ниже:

```
<ehcache:annotations>
    <ehcache:caching id="rantzCacheModel"
        cacheName="rantzCache" />
</ehcache:annotations>
```

Внутри элемента `<ehcache:annotations>` необходимо определить хотя бы один элемент `<ehcache:caching>`. Он определяет модель кеширования. Проще говоря, модель кеширования – это чуть больше, чем ссылка на именованный кеш, настроенный в файле `ehcache.xml`. В примере выше мы связали имя `rantzCacheModel` с именованным кешем `rantzCache`. Соответственно, все аннотации `@Cacheable` с атрибутом `modelId`, имеющим значение `rantzCacheModel`, будут нацелены на кеш с именем `rantzCache`.

Определение модели очистки кеша близко напоминает определение модели кеширования, за исключением того, что ссылается на очищаемый кеш. Определим модель очистки с именем `rantzFlushModel` рядом с определением модели кеширования `rantzCacheModel`, воспользовавшись элементом `<ehcache:flushing>`:

```
<ehcache:annotations>
    <ehcache:caching id="rantzCacheModel"
        cacheName="rantzCache" />
```



```
<ehcache:flushing id="rantzFlushModel"
    cacheName="rantzCache" />
</ehcache:annotations>
```

Единственное отличие модели очистки кеша от модели кеширования состоит в том, что модель очистки кеша определяет не только кеш, который будет очищаться, но и когда он будет очищаться. По умолчанию очистка кеша будет выполняться после вызова метода, отмеченного аннотацией @CacheFlush. Однако имеется возможность определить другой момент очистки, указав соответствующее значение в атрибуте `when` элемента `<ehcache:flushing>`:

```
<ehcache:annotations>
    <ehcache:caching id="rantzCacheModel"
        cacheName="rantzCache" />
    <ehcache:flushing id="rantzFlushModel"
        cacheName="rantzCache"
        when="before" />
</ehcache:annotations>
```

Значение `before` в атрибуте `when` говорит о том, что кеш будет очищаться перед вызовом метода, отмеченного аннотацией @CacheFlush.

6.7. В заключение

Данные – это кровь, текущая по жилам приложения. Некоторые из нас могут даже утверждать, что данные – это и есть приложение. При такой высокой значимости данных очень важно обеспечить надежность, простоту и ясность части приложения, отвечающей за доступ к данным.

Поддержка JDBC и фреймворков ORM принимает на себя основную тяжесть реализации доступа к данным, устранивая необходимость писать шаблонный код, присутствующий во всех механизмах хранения данных, и давая нам возможность сосредоточиться на особенностях доступа к данным, характерным для конкретного приложения.

Одним из примеров такого упрощения доступа к данным с использованием Spring является автоматическое управление жизненным циклом соединений с базами данных и сессий в фреймворках ORM, гарантирующее своевременное их открытие и закрытие. При таком подходе управление механизмами хранения данных становится практически прозрачным для прикладного кода.

Кроме того, фреймворк Spring может перехватывать исключения, специфические для используемых фреймворков (некоторые из которых могут быть контролируемыми), и преобразовывать в неконтролируемые исключения, универсальные для всех фреймворков, реализующих хранение данных, поддерживаемых фреймворком Spring. В их число входит обобщенное исключение `SQLExceptions`, возбуждаемое механизмом JDBC, которое преобразуется в более осмысленные исключения, описывающие фактическую проблему, приведшую к исключению.

В этой главе было показано, как создать слой доступа к хранилищу данных в приложениях на основе Spring с использованием JDBC, Hibernate и JPA. Выбор того или иного механизма в значительной степени зависит от личных предпочтений, но, поскольку слой доступа к данным реализуется на основе распространенных интерфейсов Java, остальная часть приложения остается независимой от используемого механизма и типа базы данных.

Еще одним важным аспектом доступа к данным, который можно упростить с помощью Spring, является управление транзакциями. В следующей главе будет показано, как можно использовать Spring AOP для декларативного управления транзакциями.



Глава 7. Управление транзакциями

В этой главе рассматриваются следующие темы:

- ❑ интеграция с диспетчерами транзакций;
- ❑ программное управление транзакциями;
- ❑ использование декларативных транзакций;
- ❑ описание транзакций с помощью аннотаций.

Отвлечемся на минутку и вспомним детство. Если вы были похожи на большинство детей, вы наверняка провели немало беззаботных часов на детской площадке, качаясь на качелях, лазая по лестницам и кружась на карусели.

Беда в том, что на качелях, устроенных в виде качающейся доски, практически невозможно качаться в одиночку. Чтобы удовольствие было полным, необходим еще один человек: вы и ваш друг оба должны согласиться покачаться на таких качелях. Такое соглашение не допускает компромиссов – все или ничего. Либо вы оба будете качаться, либо никто. Если кто-то из вас не сможет сесть на свой конец качелей, покачаться просто не получится – грустный ребенок будет сидеть на неподвижных качелях¹.

В программном обеспечении подобные бескомпромиссные операции называются *транзакциями*. Транзакции позволяют объединить несколько операций в единый блок, в котором будут выполнены либо все операции, либо ни одной. Если все идет гладко, транзакция завершается успехом. Но если что-то пойдет не так, результаты операций, которые успеют выполниться к этому моменту, будут отброшены, как если бы ничего и не происходило.

Пожалуй, самым типичным примером транзакций в реальном мире может служить перевод денег. Представьте, что вы переводите

¹ После выхода первого издания этой книги я убедился, что слово «качели» достаточно часто используется в технической литературе. Это так же просто, как окликнуть своих друзей.

\$100 со своего срочного вклада на свой же вклад до востребования. Перевод производится в виде двух последовательных операций: сначала выполняется списание \$100 со срочного вклада и затем производится зачисление этой же суммы на вклад до востребования. Перевод денег должен быть выполнен полностью или не выполнен вообще. Если операция списания будет выполнена успешно, а операция зачисления потерпит неудачу, вы потеряете \$100 (хорошо для банка, но плохо для вас). С другой стороны, если неудачу потерпит операция списания, а операция зачисления завершится успехом, вы получите лишние \$100 (хорошо для вас, но плохо для банка). Оптимальным вариантом для обеих сторон будет полный возврат вкладов к первоначальному состоянию, если какая-то операция потерпит неудачу.

В предыдущей главе мы исследовали поддержку различных механизмов доступа к данным в Spring и познакомились с несколькими способами чтения и записи данных в базу данных. При записи данных необходимо гарантировать целостность данных, выполняя запись в рамках транзакции. Фреймворк Spring обладает широкими возможностями, обеспечивая как программное, так и декларативное управление транзакциями. В этой главе будет показано, как использовать транзакции в приложениях, чтобы в случае, когда все идет как надо, результаты работы не терялись. А когда что-то пойдет не так... никто бы и не догадался об этом. (Почти никто, потому что вы как программист можете предусмотреть возможность регистрации проблем в файле журнала для дальнейшей работы над ошибками.)

7.1. Знакомство с транзакциями

Продемонстрируем применение транзакций на примере покупки билета в кино. Процесс покупки обычно включает в себя следующие операции:

- проверяется наличие свободных мест в зале;
- для каждого купленного билета количество свободных мест уменьшается на 1;
- вы оплачиваете билет;
- билет передается вам.

Если все в порядке, вы получите удовольствие от просмотра блокбастера, а кинотеатр станет на несколько долларов богаче. Но что, если случится что-то непредвиденное? Например, если во время оплаты кредитной картой выяснится, что вы превысили свой ли-

мит? Вне всяких сомнений, вы не получите билета, а кинотеатр не получит денег. Если число свободных мест не будет восстановлено в значение, предшествовавшее покупке, произойдет «утечка» свободных мест (и кинотеатр рискует недополучить прибыль). Или представьте, что может произойти, если все операции прошли успешно, но операция выдачи билета не выполнилась. Вы потеряли бы несколько долларов и довольствовались бы просмотром повторных показов по кабельному телевидению.

Чтобы гарантировать, что ни вы, ни кинотеатр ничего не потеряете, все операции должны выполняться в рамках транзакции. Так как транзакция рассматривает все заключенные в нее операции как единое действие, она гарантирует, что либо будут выполнены все операции, либо будет произведена отмена успешных выполниться операций, как если бы ничего и не произошло. Действие транзакции иллюстрируется на рис. 7.1.

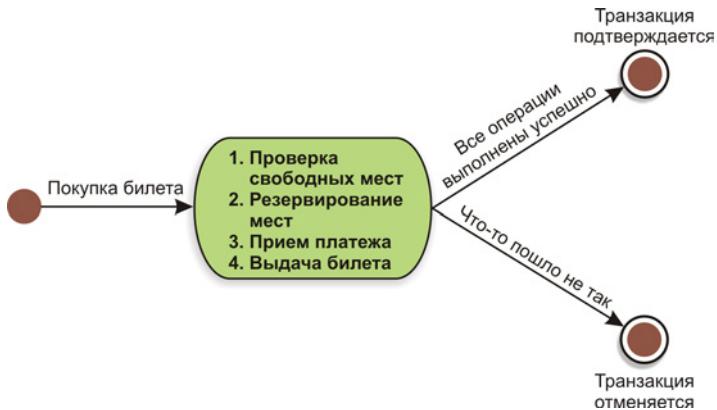


Рис. 7.1. Операции, составляющие покупку билета в кинотеатр, либо должны быть выполнены все, либо ни одна из них.

Если все операции выполнились успешно, вся транзакция завершается успехом. Иначе операции должны быть отменены, как если бы они и не выполнялись

Транзакции играют важную роль в программном обеспечении, гарантируя, что данные и ресурсы всегда будут оставаться в непротиворечивом состоянии. Без них было бы можно повредить данные или оставить их в состоянии, противоречащем логике работы приложения.

Прежде чем приступить к изучению поддержки транзакций в Spring, необходимо познакомиться с некоторыми ключевыми по-

нятиями. Рассмотрим четыре фактора, которыми руководствуются транзакции, и посмотрим, как они действуют.

7.1.1. Описание транзакций в четырех словах

В лучших традициях разработки программного обеспечения была придумана аббревиатура, описывающая транзакции: *ACID*. Эта аббревиатура происходит от следующих слов.

- ❑ *Atomic* (*атомарность*) – транзакции состоят из одной или более операций, объединенных в единицу работы. Атомарность гарантирует, что либо будут выполнены все операции, либо ни одна из них. Если все операции выполняются успешно, транзакция завершается успехом. Если какая-то операция терпит неудачу, вся транзакция терпит неудачу и отменяется.
- ❑ *Consistent* (*непротиворечивость*) – после выполнения транзакции (независимо от успеха или неудачи) система остается в состоянии, не противоречащем бизнес-модели. Данные не должны повреждаться.
- ❑ *Isolated* (*изолированность*) – транзакции должны позволять нескольким пользователям работать с одними и теми же данными, не мешая друг другу. То есть транзакции должны быть изолированы друг от друга, предотвращая возможность одновременного чтения и записи одних и тех же данных. (Обратите внимание, что изолированность обычно связана с блокировкой строк и/или таблиц в базе данных.)
- ❑ *Durable* (*долговечность*) – после выполнения транзакции результаты ее выполнения должны сохраняться, чтобы они не терялись в случае ошибок во время работы системы. Под долговечностью обычно понимается сохранение результатов в базе данных или каком-то другом хранилище.

В примере с покупкой билета под гаранцией атомарности транзакции понимается отмена всех результатов, если какая-то из операций потерпит неудачу. Атомарность обеспечивает непротиворечивость, гарантируя, что данные никогда не останутся в противоречивом, частично измененном состоянии. Изолированность также обеспечивает непротиворечивость, не позволяя другой, параллельной транзакции «украсть» у вас свободное место в ходе покупки билета.

Наконец, под долговечностью понимается сохранение результатов в некотором хранилище. В случае краха системы или других



катастрофических событий вы не должны волноваться по поводу потери результатов транзакции.

За более подробным описанием транзакций я рекомендую обратиться к книге Мартина Фаулера (Martin Fowler) «Patterns of Enterprise Application Architecture» (Addison-Wesley Professional, 2002). В частности, к главе 5, где обсуждаются транзакции и приемы параллельного доступа к данным.

Теперь, после знакомства с ключевыми особенностями транзакций, можно перейти к знакомству с поддержкой транзакций в Spring.

7.1.2. Знакомство с поддержкой транзакций в Spring

Фреймворк Spring, как и EJB, предоставляет поддержку программного и декларативного управления транзакциями. Но возможности Spring в этом отношении намного шире, чем возможности EJB.

Поддержка программного управления транзакциями в Spring существенно отличается от аналогичной ей поддержки в EJB. В отличие от EJB, где используется реализация Java Transaction API (JTA), фреймворк Spring использует механизм обратных вызовов, изолирующий фактическую реализацию транзакций от программного кода, использующего ее. В действительности поддержка управления транзакциями в Spring даже не требует наличия реализации JTA. Если приложение использует только одно хранилище данных, Spring может использовать поддержку транзакций, предлагаемую самим механизмом хранения. В число поддерживаемых механизмов входят JDBC, Hibernate и Java Persistence API (JPA). Но если требования к транзакциям в приложении распространяются на несколько хранилищ, Spring может предложить поддержку распределенных транзакций на основе сторонней реализации JTA. Подробнее поддержка программного управления транзакциями будет рассматриваться в разделе 7.3.

Программное управление транзакциями обеспечивает высочайшую гибкость и точность в определении границ транзакций, тогда как декларативное управление транзакциями (основанное на Spring AOP) помогает изолировать операции от правил применения транзакций. Поддержка декларативного управления транзакциями в Spring напоминает поддержку в EJB *транзакций, управляемых контейнером* (container-managed transactions, CMT). И та, и другая позволяют определять границы транзакций декларативно. Но

в Spring декларативное управление транзакциями предоставляет более широкие возможности, чем СМТ, позволяя объявлять дополнительные атрибуты, например определяющие уровень изоляции и пределы времени ожидания. Поддержка декларативного управления транзакциями в Spring будет рассматриваться в разделе 7.4.

Выбор между программным и декларативным управлением транзакциями в значительной степени определяется выбором между точностью управления и удобством использования. При программном управлении транзакциями приложение получает возможность точно определять границы транзакций, устанавливая начало и конец области действия транзакций. Обычно высокая точность определения границ транзакций не требуется, и поэтому чаще предпочтение отдается объявлению транзакций в файле определения контекста.

Независимо от выбранного способа управления транзакциями, программного или декларативного, в приложениях необходимо будет использовать диспетчер транзакций Spring, обеспечивающий интерфейс к конкретным реализациям транзакций. Посмотрим, насколько диспетчеры транзакций в Spring способны освободить программиста от необходимости взаимодействовать с конкретными реализациями транзакций.

7.2. Выбор диспетчера транзакций

Фреймворк Spring не осуществляет непосредственного управления транзакциями. Вместо этого в его состав входит набор *диспетчеров транзакций*, которые принимают на себя всю ответственность за управление конкретными реализациями транзакций, предоставляемых либо посредством JTA, либо механизмом хранения данных. В табл. 7.1 перечислены диспетчеры транзакций, входящие в состав Spring.

Каждый из этих диспетчеров играет роль фасада для конкретной реализации. (Взаимосвязи между некоторыми диспетчерами транзакций и конкретными реализациями изображены на рис. 7.2.) Это позволяет работать с транзакциями, не беспокоясь об особенностях каждой конкретной реализации.

Чтобы задействовать диспетчера транзакций, его необходимо объявить в контексте приложения. В данном разделе будет показано, как настраивать некоторые, наиболее широко используемые диспетчеры транзакций в Spring, начиная с диспетчера `DataSourceTransactionManager`, предоставляющего поддержку транзакций при работе с простыми механизмами хранения данных JDBC и iBATIS.

Таблица 7.1. В Spring имеется широкий выбор диспетчеров транзакций

Диспетчер транзакций (org.springframework.*.)	Область применения
jca.cci.connection. CciLocalTransactionManager	При использовании поддержки в Spring для работы с Java EE Connector Architecture (JCA) и Common Client Interface (CCI)
jdbc.datasource. DataSourceTransactionManager	Для работы с поддержкой JDBC в Spring. Также можно использовать при работе с iBATIS
jms.connection. JmsTransactionManager	При использовании JMS 1.1+
jms.connection. JmsTransactionManager102	При использовании JMS 1.0.2
orm.hibernate3. HibernateTransactionManager	При использовании Hibernate 3
orm.jdo.JdoTransactionManager	При использовании JDO
orm.jpa.JpaTransactionManager	При использовании Java Persistence API (JPA)
transaction.jta. JtaTransactionManager	При необходимости использовать распределенные транзакции или когда другие диспетчеры транзакций не соответствуют требованиям
transaction.jta. OC4JJtaTransactionManager	При использовании контейнера Oracle OC4J JEE
transaction.jta. WebLogicJtaTransactionManager	При необходимости использовать распределенные транзакции для работы с WebLogic
transaction.jta. WebSphereUowTransactionManager	При необходимости использовать транзакции, управляемые компонентом UOWManager в WebSphere

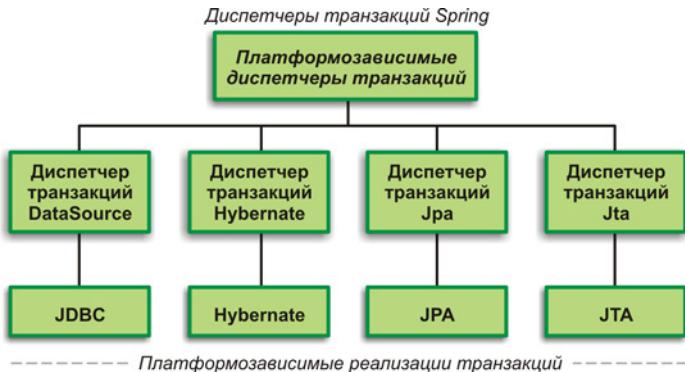


Рис. 7.2. Диспетчеры транзакций в Spring делегируют ответственность за управление транзакциями конкретным реализациям транзакций

7.2.1. Транзакции JDBC

Если для хранения данных в приложении предполагается использовать простой механизм JDBC, для управления транзакциями должен использоваться диспетчер `DataSourceTransactionManager`. Для этого необходимо добавить определение компонента `DataSourceTransactionManager` в контекст приложения, как показано ниже:

```
<bean id="transactionManager" class="org.springframework.jdbc.  
    ↳dataSource.DataSourceTransactionManager">  
    <property name="dataSource" ref="dataSource"/>  
</bean>
```

Обратите внимание, что в свойство `dataSource` записывается ссылка на компонент с именем `dataSource`. Очевидно, что компонент `dataSource` представляет реализацию интерфейса `javax.sql.DataSource` и определен где-то в файле конфигурации контекста приложения.

За кулисами компонент `DataSourceTransactionManager` управляет транзакциями, выполняя вызовы методов объекта `java.sql.Connection`, полученного из компонента `DataSource`. Например, в случае успешного выполнения транзакция подтверждается вызовом метода `commit()` объекта соединения. Аналогично, в случае неудачи, отмена транзакции производится вызовом метода `rollback()`.

7.2.2. Транзакции Hibernate

Если для доступа к хранилищу данных приложение использует фреймворк `Hibernate`, тогда должен использоваться диспетчер `HibernateTransactionManager`. При работе с версией `Hibernate 3` в определение контекста приложения необходимо добавить следующее определение элемента `<bean>`:

```
<bean id="transactionManager" class="org.springframework.  
    ↳orm.hibernate3.HibernateTransactionManager">  
    <property name="sessionFactory" ref="sessionFactory"/>  
</bean>
```

Свойство `sessionFactory` должно быть связано с компонентом типа `SessionFactory` `Hibernate`, который в данном случае имеет недвусмысленное имя `sessionFactory`. Подробности о настройке фабрики сеансов в `Hibernate` можно найти в предыдущей главе.

Что делать, если используется Hibernate 2? При использовании более старой версии механизма доступа к хранилищам данных Hibernate 2, в Spring 3.0 или Spring 2.5 не получится задействовать диспетчера `HibernateTransactionManager`. В этих версиях Spring отсутствует поддержка Hibernate 2. Если использование старой версии Hibernate является обязательным условием, придется вернуться к версии Spring 2.0.

Однако следует понимать, что при возврате к более старой версии Spring, чтобы иметь возможность пользоваться более старой версией Hibernate, теряется масса новых возможностей фреймворка Spring, обсуждаемых в этой книге. Поэтому, прежде чем откатываться к более старой версии Spring, я рекомендую попробовать обновить версию Hibernate.

Диспетчер `HibernateTransactionManager` возлагает всю ответственность за управление транзакциями на объект `org.hibernate.Transaction`, который он получает из объекта сеанса Hibernate. В случае успешного выполнения транзакция подтверждается вызовом метода `commit()` объекта `Transaction`. Аналогично, в случае неудачи, отмена транзакции производится вызовом метода `rollback()` объекта `Transaction`.

7.2.3. Транзакции Java Persistence API

Фреймворк Hibernate уже в течение многих лет фактически является стандартным механизмом хранения данных в Java-приложениях, но совсем недавно на сцену вышла библиотека Java Persistence API (JPA), ставшая действительным стандартом в области хранения данных. Если вы готовы перейти на использование JPA, тогда для управления транзакциями вам потребуется диспетчер `JpaTransactionManager`. Ниже показано, как выполняется настройка компонента `JpaTransactionManager` в Spring:

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

Диспетчеру `JpaTransactionManager` необходима только ссылка на фабрику диспетчера сущностей JPA (на любую реализацию интерфейса `javax.persistence.EntityManagerFactory`). Управление транзакциями `JpaTransactionManager` будет осуществляться посредством объекта `EntityManager`, возвращаемого фабрикой.

Помимо применения транзакций к операциям JPA, диспетчер `JpaTransactionManager` также поддерживает транзакции для простых

операций JDBC в том же самом источнике данных DataSource, используемом фабрикой EntityManagerFactory. Чтобы воспользоваться этой поддержкой, диспетчера JpaTransactionManager необходимо также связать с реализацией интерфейса JpaDialect. Например, предположим, что в приложении используется компонент EclipseLinkJpaDialect, настроенный следующим образом:

```
<bean id="jpaDialect"
      class="org.springframework.orm.jpa.vendor.EclipseLinkJpaDialect" />
```

Тогда вам необходимо внедрить компонент jpaDialect в компонент JpaTransactionManager, как показано ниже:

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

Важно отметить, что при этом реализация JpaDialect должна поддерживать смешанный JPA/JDBC доступ к данным. Все реализации JpaDialect (EclipseLinkJpaDialect, HibernateJpaDialect, OpenJpaDialect и TopLinkJpaDialect), входящие в состав Spring, обеспечивают такую смешанную поддержку механизмов JPA и JDBC. А реализация DefaultJpaDialect – нет.

7.2.4. Транзакции Java Transaction API

Если ни один из вышеупомянутых диспетчеров транзакций не отвечает предъявляемым требованиям или если необходимо охватить транзакциями несколько источников данных (например, две или более различные базы данных), можно воспользоваться диспетчером JtaTransactionManager:

```
<bean id="transactionManager" class="org.springframework.
  ↗transaction.jta.JtaTransactionManager">
  <property name="transactionManagerName"
    value="java:/TransactionManager" />
</bean>
```

Диспетчер JtaTransactionManager возлагает всю ответственность за управление транзакциями на реализацию а JTA. JTA определяет



стандартный API для управления транзакциями, охватывающими один или более источников данных. Свойство `transactionManagerName` определяет имя диспетчера транзакций в JNDI.

`JtaTransactionManager` действует совместно с объектами `javax.transaction.UserTransaction` и `javax.transaction.TransactionManager`, делегируя им ответственность за управление транзакциями. В случае успешного выполнения транзакция подтверждается вызовом метода `UserTransaction.commit()`. Аналогично, в случае неудачи, отмена транзакции производится вызовом метода `UserTransaction.rollback()`.

На данный момент совершенно очевидно, что диспетчеры транзакций Spring лучше всего соответствуют потребностям приложения Spitter, поскольку для хранения данных в нем был выбран один из поддерживаемых механизмов хранения данных. Теперь пришло время задействовать диспетчера транзакций. Начнем с реализации программного управления транзакциями вручную.

7.3. Программное управление транзакциями в Spring

Существуют два типа людей: любители покомандовать и те, кто командовать не любят. Любители покомандовать стремятся контролировать все и вся, и никому не доверяют. Если вы – разработчик и любите покомандовать, вы наверняка относитесь к числу людей, предпочитающих командную строку, и пишете свои методы доступа к свойствам, не доверяя эту работу среде разработки.

Любителям командовать также нравится точно знать, что происходит в их программном коде. Когда дело доходит до транзакций, они предпочитают полностью контролировать точку запуска транзакции, точку ее подтверждения и завершения. Декларативное управление транзакциями оказывается недостаточно точным инструментом для них.

Впрочем, это не так уж и плохо. Любители покомандовать отчасти правы. Как будет показано ниже в этой главе, декларативное управление транзакциями ограничивается уровнем методов. Если потребуется более точное управление границами транзакций, этого можно будет добиться, используя только программное управление транзакциями.

В качестве примера метода, выполняемого в рамках транзакции, возьмем метод `saveSpittle()` класса `SpitterServiceImpl`, представленный в листинге 7.1.

Листинг 7.1. Метод saveSpittle() сохраняет объект Spittle

```
public void saveSpittle(Spittle spittle) {  
    spitterDao.saveSpittle(spittle);  
}
```

Несмотря на кажущуюся простоту, в этом методе могут выполняться довольно сложные операции, невидимые на первый взгляд. В процессе сохранения объекта Spittle механизм хранения данных может выполнять множество действий. Даже если сохранение сводится к простой вставке строки в таблицу базы данных, важно убедиться, что все операции выполняются в рамках транзакции. Если все операции будут выполнены успешно, транзакцию можно подтвердить. Если что-то пойдет не так, ее можно отменить.

Один из подходов к включению транзакций в работу заключается в добавлении транзакции программно, непосредственно внутри метода saveSpittle(), с использованием класса шаблона TransactionTemplate. Подобно другим классам шаблонов в Spring (таким как класс JdbcTemplate, обсуждавшийся в предыдущей главе), TransactionTemplate использует механизм обратных вызовов. В листинге 7.2 приводится измененная версия метода saveSpittle(), демонстрирующая, как добавлять транзакции с помощью TransactionTemplate.

Листинг 7.2. Программное добавление транзакции в метод saveSpittle()

```
public void saveSpittle(final Spittle spittle) {  
    txTemplate.execute(new TransactionCallback<Void>() {  
        public Void doInTransaction(TransactionStatus txStatus) {  
            try {  
                spitterDao.saveSpittle(spittle);  
            } catch (RuntimeException e) {  
                txStatus.setRollbackOnly();  
                throw e;  
            }  
            return null;  
        }  
    });  
}
```

Чтобы иметь возможность использовать класс TransactionTemplate, необходимо сначала реализовать интерфейс TransactionCallback. Так как интерфейс TransactionCallback определяет единственный метод, часто бывает проще реализовать его в виде анонимного вложенного

класса, как показано в листинге 7.2. А программный код, который должен выполняться в рамках транзакции, должен находиться внутри метода `doInTransaction()`.

Вызов метода `execute()` экземпляра класса `TransactionTemplate` выполнит программный код внутри экземпляра `TransactionCallback`. В случае появления проблемы будет вызван метод `setRollbackOnly()` объекта `TransactionStatus`, чтобы отменить транзакцию. В противном случае, если метод `doInTransaction()` благополучно вернется управление, транзакция будет подтверждена.

А откуда здесь возьмется экземпляр `TransactionTemplate`? Хороший вопрос. Он должен быть внедрен в компонент `SpitterServiceImpl`, как показано ниже:

```
<bean id="spitterService"
      class="com.habuma.spitter.service.SpitterServiceImpl">
    ...
    <property name="transactionTemplate ">
      <bean class="org.springframework.transaction.support.
                  TransactionTemplate">
        <property name="transactionManager"
                  ref="transactionManager" />
      </bean>
    </property>
</bean>
```

Обратите внимание на наличие в компоненте `TransactionTemplate` свойства `transactionManager`. За кулисами компонент `TransactionTemplate` использует реализацию интерфейса `PlatformTransactionManager` для обслуживания особенностей транзакций для конкретной платформы. Здесь в свойство внедряется ссылка на компонент с именем `transactionManager`, который может быть любым диспетчером транзакций из числа перечисленных в табл. 7.1.

Программное управление транзакциями отлично подходит для ситуаций, когда требуется иметь полный контроль над границами транзакций. Но, как видно из листинга 7.2, это достаточно утомительно. Необходимо изменить реализацию метода `saveSpittle()` – использовать классы из фреймворка Spring, чтобы ввести в действие поддержку программного управления транзакциями в Spring.

Чаще всего в приложениях не требуется такая точность управления границами транзакций. Именно по этой причине обычно предпочитают объявлять транзакции за пределами программного кода

(в конфигурационном файле Spring, например). В оставшейся части главы будет рассматриваться поддержка в Spring декларативного управления транзакциями.

7.4. Декларативное управление транзакциями

Еще совсем недавно *декларативное управление транзакциями* было доступно только в контейнерах EJB. Но сейчас Spring предлагает аналогичную поддержку для POJO. Это – важная особенность Spring, потому что теперь для декларативного обеспечения атомарности операций не требуется создавать контейнеры EJB.

Поддержка декларативного управления транзакциями в Spring реализована посредством фреймворка Spring AOP. Это вполне естественно, потому что транзакции – это системная служба, стоящая уровнем выше приложения. Транзакции в Spring можно интерпретировать как аспекты, «обертывающие» методы.

Фреймворк Spring предоставляет три способа объявления границ транзакций. Исторически фреймворк Spring всегда обладал поддержкой декларативного управления транзакциями за счет проксирования компонентов с использованием Spring AOP и `TransactionProxyFactoryBean`. Но, начиная с версии Spring 2.0, появился более удобный способ объявления транзакций, основанный на использовании конфигурационного пространства имен `tx` и аннотации `@Transactional`.

Несмотря на то что класс `TransactionProxyFactoryBean` все еще доступен в современных версиях Spring, он считается устаревшим и потому не будет рассматриваться здесь. Вместо этого мы сконцентрируемся на использовании пространства имен `tx` и объявлении транзакций с помощью аннотации. Но сначала исследуем определение атрибутов транзакций.

7.4.1. Определение атрибутов транзакций

В Spring декларативные транзакции определяются с помощью *атрибутов транзакций*. Атрибуты транзакции – это описание особенностей применения транзакции к методу. Всего имеется пять различных атрибутов транзакции, как показано на рис. 7.3.

Фреймворк Spring предоставляет несколько механизмов объявления транзакций, однако все они опираются на эти пять параметров,



Рис. 7.3. Декларативные транзакции определяются в терминах правил распространения, уровня изоляции, признака «только для чтения», предельного времени ожидания и правил отмены

управляющих поведением транзакций. Поэтому, чтобы научиться объявлять транзакции в Spring, важно разобраться с этими параметрами.

Независимо от используемого механизма декларативных транзакций у вас всегда будет возможность определить данные атрибуты. Исследуем каждый атрибут отдельно и посмотрим, какое влияние они оказывают на транзакции.

Правила распространения

Первый атрибут, который мы рассмотрим, определяет *правила распространения* транзакции. Этот атрибут назначает границы транзакции. Всего фреймворк Spring определяет семь разных правил распространения, которые перечислены в табл. 7.2.

Константы, описывающие правила распространения. Правила распространения транзакций, перечисленные в табл. 7.2, определены в виде констант в интерфейсе `org.springframework.transaction.TransactionDefinition`.

Правила распространения транзакций, перечисленные в табл. 7.2, кому-то могут показаться знакомыми. И это неудивительно, потому что они отражают правила распространения транзакций, управляемых контейнером, в EJB (*container-managed transactions*, CMT). Например, правило `PROPAGATION_REQUIRE_NEW` в Spring эквивалентно правилу `RequiresNew` в CMT. Однако в Spring добавлено одно дополнительное правило распространения, отсутствующее в CMT, – `PROPAGATION_NESTED`, обеспечивающее поддержку вложенных транзакций.

Правила распространения отвечают на вопрос: «должна ли быть запущена новая или приостановлена выполняющаяся транзакция» или «должен ли метод выполняться в контексте транзакции».

Таблица 7.2. Правила распространения определяют, когда будет запущена новая транзакция, а когда использоваться имеющаяся. Фреймворк Spring предлагает на выбор несколько правил

Правило распространения	Описание
PROPAGATION_MANDATORY	Указывает, что метод должен выполняться внутри транзакции. Если к моменту вызова метода не будет запущена транзакция, фреймворк возбудит исключение
PROPAGATION_NESTED	Указывает, что метод должен выполняться внутри вложенной транзакции, если к моменту вызова метода уже была запущена транзакция. Вложенную транзакцию можно подтвердить или отменить независимо от вмещающей транзакции. При отсутствии вмещающей транзакции это правило действует подобно правилу PROPAGATION_REQUIRED. Поддержка этого правила различными фреймворками хранения данных в лучшем случае является неполной. Чтобы выяснить наличие поддержки вложенных транзакций используемым фреймворком, обращайтесь к его документации
PROPAGATION_NEVER	Указывает, что метод не должен выполнятся внутри транзакции. Если к моменту вызова метода будет запущена транзакция, фреймворк возбудит исключение
PROPAGATION_NOT_SUPPORTED	Указывает, что метод не должен выполняться внутри транзакции. Если к моменту вызова метода будет запущена транзакция, фреймворк приостановит ее на время выполнения метода. При использовании JTATransactionManager необходим доступ к TransactionManager
PROPAGATION_REQUIRED	Указывает, что метод должен выполняться внутри транзакции. Если к моменту вызова метода будет запущена транзакция, он будет выполняться в рамках этой транзакции. В противном случае будет запущена новая транзакция
PROPAGATIONQUIRES_NEW	Указывает, что метод должен выполнятся внутри собственной транзакции. Будет запущена новая транзакция, и, если к моменту вызова метода уже будет запущена другая транзакция, фреймворк приостановит ее на время выполнения метода. При использовании JTATransactionManager необходим доступ к TransactionManager

**Таблица 7.2 (окончание)**

Правило распространения	Описание
PROPAGATION_SUPPORTS	Указывает, что метод не требует наличия транзакции, но он может выполняться внутри имеющейся транзакции

Например, если метод объявлен как выполняющийся в рамках транзакции с правилом PROPAGATION.Requires_New, это означает, что границы транзакции совпадают с границами метода: перед вызовом метода запускается новая транзакция, а когда метод возвращает управление или возбуждает исключение, транзакция завершается. В случае использования правила PROPAGATION_REQUIRED границы транзакции будут зависеть от того, была ли запущена другая транзакция.

Уровни изоляции

Второй атрибут, описывающий поведение декларативной транзакции, определяет *уровень изоляции*. Уровень изоляции указывает, насколько транзакция подвержена влиянию других транзакций, выполняющихся параллельно. Уровень изоляции транзакции можно также представить как меру эгоизма транзакции по отношению к данным, используемым в рамках транзакции.

В типичных приложениях транзакции, выполняющиеся параллельно, часто работают с одними и теми же данными. Параллельное выполнение транзакций может привести к следующим проблемам.

- ❑ *Чтение неподтвержденных данных* – когда одна транзакция читает данные, записанные, но не подтвержденные другой транзакцией. Если позднее другая транзакция будет отменена, данные, полученные первой транзакцией, окажутся недействительными.
- ❑ *Неповторимость получаемых результатов* – когда транзакция несколько раз выполняет один и тот же запрос и каждый раз получает разные данные. Обычно это обусловлено действием других транзакций, успевающих внести изменения между выполнением запросов.
- ❑ *Чтение фантомных данных* – напоминает неповторимость получаемых результатов. Эта ситуация может возникать, когда транзакция (T1) читает несколько строк, а затем вторая транзакция (T2), выполняющаяся параллельно, вставляет несколько строк. При выполнении последующих запросов первая транзакция (T1) будет обнаруживать дополнительные строки, отсутствовавшие прежде.

В идеале, чтобы избежать подобных проблем, транзакции должны быть полностью изолированы друг от друга. Но полная изоляция может отрицательно сказаться на производительности, потому что часто для этого необходимо блокировать доступ к строкам (а иногда и к таблицам целиком) в хранилище данных. Агрессивное использование блокировок может воспрепятствовать параллельному выполнению транзакций, вынуждая их ждать, пока другие транзакции не выполнят свою работу.

Учитывая, что полная изоляция может влиять на производительность и она необходима далеко не во всех приложениях, иногда бывает желательно иметь более гибкую возможность изолирования транзакций. Поэтому транзакции поддерживают несколько уровней изоляции, перечисленные в табл. 7.3.

Таблица 7.3. Уровни изоляции определяют, до какой степени транзакции могут влиять друг на друга при одновременном выполнении

Уровень изоляции	Описание
ISOLATION_DEFAULT	Используется уровень изоляции по умолчанию, зависящий от конкретного хранилища данных
ISOLATION_READ_UNCOMMITTED	Позволяет читать неподтвержденные изменения. Может привести к проблеме чтения неподтвержденных данных, неповторимости получаемых результатов и чтения фантомных данных
ISOLATION_READ_COMMITTED	Позволяет читать подтвержденные изменения, выполненные в других, параллельно выполняющихся транзакциях. Устраняет проблему чтения неподтвержденных данных, но проблемы неповторимости получаемых результатов и чтения фантомных данных все еще могут возникать
ISOLATION_REPEATABLE_READ	Многократные попытки чтения одного и того же поля будут возвращать одни и те же результаты, если эти поля не будут изменяться в пределах самой транзакции. Устраниет проблемы чтения неподтвержденных данных и неповторимости получаемых результатов, но проблема чтения фантомных данных все еще может возникать
ISOLATION_SERIALIZABLE	Этот уровень изоляции полностью соответствует определению ACID и устраниет проблемы чтения неподтвержденных данных, неповторимости получаемых результатов и чтения фантомных данных. Этот уровень изоляции снижает производительность приложения в большей степени, чем все остальные, потому что такая изоляция достигается за счет полного блокирования доступа к таблицам, используемым в транзакции

Константы, описывающие уровни изоляции. Уровни изоляций, перечисленные в табл. 7.3, определены в виде констант в интерфейсе `org.springframework.transaction.TransactionDefinition`.

Наибольшую производительность обеспечивает уровень изоляции `ISOLATION_READ_UNCOMMITTED`, но он в наименьшей степени изолирует транзакции друг от друга, оставляя возможность появления проблем чтения неподтвержденных данных, неповторимости получаемых результатов и чтения фантомных данных. Полной его противоположностью является уровень изоляции `ISOLATION_SERIALIZABLE`, предотвращающий все эти проблемы, но он оказывает самое сильное отрицательное влияние на производительность.

Помните, что не все источники данных поддерживают все уровни изоляции, перечисленные в табл. 7.3. Обязательно обращайтесь к соответствующей документации, чтобы выяснить, какие уровни изоляции поддерживаются.

Только для чтения

Третим параметром объявления транзакции является *признак доступа к данным только для чтения*. На тот случай, если транзакция будет выполнять только операции чтения данных, хранилище может предусматривать некоторые оптимизации, учитывающие природу таких транзакций. Объявляя транзакцию как выполняющую только чтение данных, вы даете хранилищу данных возможность применить эти оптимизации.

Поскольку оптимизация доступа к данным только для чтения выполняется хранилищем данных в начале транзакции, этот признак имеет смысл использовать лишь при объявлении транзакций для методов, для которых правило распространения допускает создание новой транзакции (`PROPAGATION_REQUIRED`, `PROPAGATION.Requires_NEW` и `PROPAGATION_NESTED`).

Кроме того, при использовании механизма хранения данных Hibernate объявление транзакции как выполняющей только операции чтения, приведет к тому, что в Hibernate режим выталкивания получит значение `FLUSH_NEVER`, предписывающее фреймворку Hibernate избегать ненужной синхронизации объектов с базой данных, откладывая все обновления до конца транзакции.

Время ожидания

При нормальной работе приложения транзакции не могут выполняться в течение длительного времени. Поэтому следующей харак-

теристикой в объявлениях транзакций является параметр, определяющий время ожидания.

Представьте, что транзакция неожиданно стала выполняться слишком долго. Поскольку выполнение транзакций может блокироваться хранилищем данных, долго выполняющиеся транзакции могут удерживать ресурсы базы данных неоправданно продолжительное время. Вместо того чтобы ждать, можно объявить, что транзакция должна автоматически отменять выполненные в ней операции по прошествии определенного количества секунд.

Поскольку отсчет времени ожидания начинается с момента запуска транзакции, объявлять предельное время ожидания имеет смысл только при объявлении транзакций для методов, для которых правило распространения допускает создание новой транзакции (`PROPAGATION_REQUIRED`, `PROPAGATIONQUIRES_NEW` и `PROPAGATION_NESTED`).

Правила отмены

Последней гранью пятиугольника, изображенного на рис. 7.3, является набор правил, определяющих, какие исключения должны приводить к отмене транзакции, а какие нет. По умолчанию транзакции отменяются только в случае появления исключений времени выполнения, но не в случае контролируемых исключений (*checked exceptions*). (Это соответствует правилам отмены, применяемым в EJB.)

Однако есть возможность объявить, что транзакция должна отменяться в ответ не только на исключения времени выполнения, но и на определенные контролируемые исключения. Аналогично можно объявить, что транзакция не должна отменяться в ответ на указанные исключения, даже если они являются исключениями времени выполнения.

Теперь, после знакомства с атрибутами транзакций, описывающими их поведение, можно посмотреть, как использовать эти атрибуты при объявлении транзакций в Spring.

7.4.2. Объявление транзакций в XML

В предыдущих версиях Spring объявление транзакций было связано с внедрением специального компонента `TransactionProxyFactoryBean`. Проблема с компонентом `TransactionProxyFactoryBean` состоит в том, что его использование ведет к разбуханию конфигурационных файлов Spring. К счастью, в настоящее время эта проблема была устранена, и теперь фреймворк Spring предлагает конфигурацион-

ное пространство имен tx, существенно упрощающее использование декларативных транзакций в Spring.

Чтобы использовать пространство имен tx, необходимо добавить его в конфигурационный XML-файл Spring:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-3.0.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
```

Обратите внимание, что при этом также необходимо включить пространство имен aop. Это очень важно, потому что некоторые элементы настройки декларативных транзакций опираются на отдельные элементы настройки AOP (представленные в главе 5).

Пространство имен tx содержит несколько новых конфигурационных XML-элементов, наиболее примечательным из которых является `<tx:advice>`. Следующий фрагмент XML-файла демонстрирует, как можно использовать элемент `<tx:advice>` для объявления транзакции, подобной той, что была определена для службы Spitter в листинге 7.2:

```
<tx:advice id="txAdvice">
    <tx:attributes>
        <tx:method name="add*" propagation="REQUIRED" />
        <tx:method name="*" propagation="SUPPORTS"
            read-only="true"/>
    </tx:attributes>
</tx:advice>
```

Атрибуты транзакции внутри элемента `<tx:advice>` определяются элементом `<tx:attributes>` с помощью одного или более элементов `<tx:method>`. Элемент `<tx:method>` определяет атрибуты транзакции для метода (или методов), указанного в атрибуте `name` (допускается использование шаблонных символов).

Элемент `<tx:method>` имеет несколько атрибутов, помогающих определять поведение транзакции, которые перечислены в табл. 7.4.

Таблица 7.4. Пять сторон пятиугольника поведения транзакции (рис. 7.3) определяются атрибутами элемента <tx:method>

Атрибут	Описание
isolation	Определяет уровень изоляции транзакции
propagation	Определяет правила распространения транзакции
read-only	Определяет режим доступа к данным
Правила отмены: rollback-for no-rollback-for	rollback-for определяет проверяемые исключения, при появлении которых транзакция должна отменяться. no-rollback-for определяет исключения, при появлении которых транзакция не должна отменяться
timeout	Определяет предельное время ожидания для транзакций, выполняющихся продолжительное время

Согласно определению совета txAdvice транзакции, методы, на которые распространяется эта транзакция, делятся на две категории: имена которых начинаются с add и все остальные. Метод saveSpittle() попадает в первую категорию и объявляется как обязательно выполняющийся в рамках транзакции. Остальные методы определяются с правилом propagation="supports" – они не требуют выполнения в рамках транзакции, только если таковая существует.

При объявлении транзакции с помощью элемента <tx:advice> все еще необходим диспетчер транзакций, как при использовании компонента TransactionProxyFactoryBean. В случае с элементом <tx:advice> предполагается, что диспетчер транзакций будет объявлен как компонент с идентификатором transactionManager. Если в конфигурации диспетчера транзакций присвоено другое имя (например, txManager), необходимо указать этот идентификатор в атрибуте transactionManager:

```
<tx:advice id="txAdvice">
    transaction-manager="txManager">
    ...
</tx:advice>
```

Сам по себе элемент <tx:advice> определяет только совет аспекта для применения к методам, требующим выполнения в рамках транзакций. Но это лишь совет, а не полноценный аспект. Нигде в элементе <tx:advice> не указывается, к какому компоненту применяется этот совет, – для этого необходимо определить срез множества точек внедрения. Чтобы создать полное определение аспекта транзакции, следует определить объект-советник. Именно для этого

и необходимо пространство имен aop. Следующий фрагмент XML-файла определяет объект-советник, использующий совет txAdvice для применения к любым компонентам, реализующим интерфейс SpitterService:

```
<aop:config>
    <aop:advisor
        pointcut="execution(* *..SpitterService.*(..))"
        advice-ref="txAdvice"/>
</aop:config>
```

В атрибуте pointcut используется выражение AspectJ выборки точек внедрения, указывающее, что этот объект-советник должен применить совет ко всем методам интерфейса SpitterService. Перечень методов, выполняющихся в рамках транзакции, а также атрибуты транзакции определяются самим советом txAdvice, на который ссылается атрибут advice-ref совета.

Несмотря на то что элемент `<tx:advice>` несет немало удобств в создании декларативных транзакций, тем не менее в Spring 2 имеется еще одна возможность, которая выглядит еще более привлекательной для тех, кто работает в окружении Java 5. Посмотрим, как можно управлять транзакциями с помощью аннотаций.

7.4.3. Определение транзакций с помощью аннотаций

Применение элемента `<tx:advice>` позволяет значительно упростить объявление транзакций в конфигурационных XML-файлах Spring. А что, если я скажу, что можно добиться еще большего упрощения? Что, если я скажу, что достаточно добавить в определение контекста приложения всего одну строку, чтобы обеспечить объявление транзакций?

Помимо элемента `<tx:advice>`, пространство имен tx содержит также элемент `<tx:annotation-driven>`. Чтобы воспользоваться им, достаточно добавить в XML-файл всего одну строку:

```
<tx:annotation-driven />
```

И все! Если вы ожидали большего, прошу свои извинения. Чтобы сделать пример немного интереснее, можно добавить определение компонента диспетчера транзакций с помощью атрибута

transactionmanager (который по умолчанию ссылается на компонент с идентификатором transactionManager):

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

Но это все, что можно добавить. Эта короткая строка в XML-файле обладает большими возможностями, позволяя определять транзакции там, где это более уместно: в методах, которые должны выполняться в рамках транзакций.

Аннотации – одна из самых крупных и самых обсуждаемых особенностей Java 5. Аннотации позволяют добавлять метаданные непосредственно в программный код, а не во внешние конфигурационные файлы. Я полагаю, что они прекрасно подходят для объявления транзакций.

Элемент `<tx:annotation-driven>` сообщает фреймворку Spring проверить все компоненты в контексте приложения и отыскать отмеченные аннотацией `@Transactional` на уровне определения класса или на уровне методов. К каждому компоненту с аннотацией `@Transactional` элемент `<tx:annotation-driven>` автоматически применит совет с определением транзакции. Атрибуты транзакции в этом случае определяются параметрами аннотации `@Transactional`.

Например, в листинге 7.3 демонстрируется объявление класса `SpitterServiceImpl`, дополненное аннотациями `@Transactional`.

Листинг 7.3. Добавление транзакций в службу Spitter

```
@Transactional(propagation=Propagation.SUPPORTS, readOnly=true)
public class SpitterServiceImpl implements SpitterService {
    ...
    @Transactional(propagation=Propagation.REQUIRED, readOnly=false)
    public void addSpitter(Spitter spitter) {
    ...
    }
}
```

Аннотация `@Transactional` на уровне определения класса `SpitterServiceImpl` сообщает, что все его методы поддерживают возможность выполнения в рамках транзакций, выполняющихся в режиме «только для чтения». А аннотация на уровне метода `saveSpittle()` сообщает, что этот метод обязательно должен выполняться в контексте транзакции.



7.5. В заключение

Транзакции занимают важное место в корпоративных приложениях, повышая их надежность. Они обеспечивают бескомпромиссность операций, предотвращая повреждение данных в случае непредвиденных событий. Они также поддерживают возможность параллельного выполнения, исключая возможность взаимовлияния потоков выполнения в многопоточных приложениях, оперирующих одними и теми же данными.

Фреймворк Spring поддерживает и программное, и декларативное управление транзакциями. В обоих случаях Spring ограждает вас от необходимости работать непосредственно с конкретными механизмами управления транзакциями, скрывая конкретные реализации за универсальным API.

Для поддержки декларативного управления транзакциями Spring использует собственный фреймворк АОР. Декларативные транзакции в Spring обладают теми же возможностями, что и конкурирующая платформа EJB СМТ, позволяя определять не только правила распространения в РОJO, но также уровень изоляции, задействовать оптимизации, подразумеваемые режимом доступа «только для чтения», и правила отмены в ответ на конкретные исключения.

Эта глава продемонстрировала, как применять декларативные транзакции с использованием аннотаций модели программирования Java 5. С введением аннотаций в Java 5, чтобы обеспечить выполнение метода в рамках транзакции, достаточно просто пометить его соответствующей аннотацией.

Как было показано выше, Spring распространяет мощь декларативных транзакций на РОJO. Это самое захватывающее нововведение, так как прежде декларативные транзакции были доступны только в EJB. Но декларативные транзакции – это лишь верхушка айсберга из всего того, что Spring может предложить для работы с РОJO. В следующей главе будет показано, как Spring распространяет на РОJO декларативную поддержку безопасности.



Глава 8. Создание веб-приложений с помощью Spring MVC

В этой главе рассматриваются следующие темы:

- отображение запросов в контроллеры Spring;
- прозрачное связывание параметров форм;
- проверка заполнения присылаемых форм;
- выгрузка файлов.

Будучи разработчиком корпоративных приложений на языке Java, вам наверняка приходилось заниматься разработкой веб-приложений. Для многих разработчиков на Java создание веб-приложений является основным занятием. Если у вас есть подобный опыт, вы наверняка хорошо осведомлены о характерных проблемах, которые приходится решать в подобных системах, таких как поддержка информации о текущем состоянии или проверка входных данных. И все они усугубляются еще больше независимой природой протокола HTTP.

С целью помочь в решении всех этих проблем был разработан веб-фреймворк, входящий в состав Spring. Опираясь на шаблон модель–представление–контроллер (Model-View-Controller, MVC), фреймворк Spring MVC помогает строить веб-приложения, столь же гибкие и слабо связанные, как сам фреймворк Spring Framework.

В этой главе мы займемся изучением веб-фреймворка Spring MVC и новых аннотаций, сконструируем контроллеры, обрабатывающие веб-запросы. В процессе разработки мы будем стремиться проектировать веб-слой приложения в соответствии с архитектурой RESTful. В заключение посмотрим, как использовать теги JSP в представлениях, отправляемых пользователям в ответ на запросы.

Однако прежде чем углубляться в особенности реализации контроллеров и отображения запросов, окнем беглым взглядом фреймворк Spring MVC и сконструируем простенькое веб-приложение.

8.1. Обзор Spring MVC

Видели ли вы когда нибудь детскую игру «Мышеловка»? Довольно забавная игра. Ее цель – провести небольшой стальной шарик в мышеловку через ряд хитрых приспособлений. Шарик катится через различные сложные конструкции, сначала по извилистой дорожке через игрушечные качели на миниатюрное колесо обозрения, а потом подбрасывается из крошечного ведерка ударом резинового башмачка. И все только чтобы в конце защелкнуть в ловушке бедную, ничего не подозревающую пластмассовую мышку.

На первый взгляд создается ощущение, что фреймворк MVC Spring напоминает игру «Мышеловка». Только вместо перемещения шарика через различные желобки, качели и колеса Spring запутанными кругами перемещает запросы между сервлетом-диспетчером, механизмами отображения, контроллерами и представлениями.

Но не стоит слишком отождествлять Spring MVC с игрой «Мышеловка» в стиле Руби Голдберга. Каждый из компонентов в Spring MVC выполняет вполне конкретную задачу. Начнем изучение Spring MVC, ознакомившись с жизненным циклом типичного запроса.

8.1.1. Путь одного запроса через Spring MVC

Каждый раз, когда пользователь щелкает на ссылке или отправляет форму в веб-браузере, запрос отправляется на работу. Наш запрос работает курьером. Подобно почтальону или курьеру из службы доставки, запрос переносит информацию из одного места в другое.

Запрос – весьма занятой парень. С момента, когда он покинет браузер, и до момента, когда вернется ответ, запрос сделает несколько остановок, каждый раз сбрасывая часть информации и подбирая что-то взамен. На рис. 8.1 показаны все остановки, которые делает запрос.

Когда запрос покидает браузер, он несет в себе информацию о требовании пользователя. По крайней мере, запрос будет нести в себе запрошенный URL. Но он может также нести дополнительные данные, такие как информация из формы, заполненной пользователем.

Первой остановкой на пути запроса является `DispatcherServlet`. Как и большинство веб-фреймворков на языке Java, фреймворк Spring MVC пропускает все входящие запросы через единственный сервlet входного контроллера. *Входной контроллер* (*front controller*) является типичным шаблоном проектирования веб-приложений, где



Рис. 8.1. Веб-слой приложения Spitter включает два контроллера ресурсов наряду с несколькими вспомогательными контроллерами

единственный сервлет берет на себя ответственность за передачу всех запросов остальным компонентам приложения, выполняющим фактическую их обработку. В Spring MVC входным контроллером является `DispatcherServlet`.

Задача контроллера `DispatcherServlet` состоит в том, чтобы передать запрос контроллеру Spring MVC. Контроллер – это компонент Spring, обрабатывающий запрос. Но приложение может иметь несколько контроллеров, и входному контроллеру `DispatcherServlet` требуется помочь, чтобы определить, какому контроллеру передать запрос. Поэтому контроллер `DispatcherServlet` консультируется с одним или несколькими механизмами отображения и выясняет, где будет следующая остановка запроса. При принятии решения механизм отображения в первую очередь руководствуется адресом URL в запросе.

Как только будет выбран соответствующий контроллер, `DispatcherServlet` отправляет запрос в путь к выбранному контроллеру. Достигнув контроллера, запрос отдаст часть своего груза (информацию, отправленную пользователем) и терпеливо будет ждать, пока контроллер обработает эту информацию. (На самом деле хорошо спроектированный контроллер сам почти не занимается обработкой информации, вместо этого он делегирует ответственность за обработку одному или нескольким служебным объектам.)

В результате работы контроллера часто появляется некоторая информация, которая должна быть передана назад пользователю и отображена в браузере. Эта информация называется *моделью*. Но

отправки обратно необработанной информации недостаточно, перед отправкой ее следует представить в удобном для пользователя формате, обычно в HTML. Для этого информация должна быть передана в одно из *представлений*, которыми обычно являются JSP.

Последнее, что должен сделать контроллер, – упаковать вместе модель и имя представления для отображения результатов в браузере. Затем он отсылает запрос вместе с моделью и именем представления обратно входному контроллеру DispatcherServlet.

Чтобы контроллер не оказался тесно связанным с каким-либо конкретным представлением, имя представления, возвращаемое входному контроллеру DispatcherServlet, не определяет JSP-страницу непосредственно. Фактически оно даже не предполагает, что представление вообще является страницей JSP. Оно является лишь логическим именем представления, используемым затем для поиска фактического представления. Чтобы отобразить логическое имя представления в ссылку на конкретную реализацию, входной контроллер DispatcherServlet обратится к арбитру представлений (view resolver).

Теперь, когда контроллер DispatcherServlet определил, какое представление будет отображать результаты, работа запроса подошла к концу. Его конечная остановка – реализация представления (возможно, страница JSP), куда он доставит модель данных. На этом работа запроса заканчивается. На основе модели данных представление создаст отображение страницы, которое будет отправлено обратно клиенту с другим (не таким трудолюбивым) курьером – объектом ответа.

В этой главе мы подробно рассмотрим каждый из этих этапов. Но сначала необходимо настроить Spring MVC и компонент DispatcherServlet.

8.1.2. Настройка Spring MVC

Основой Spring MVC является сервлет DispatcherServlet, который играет роль входного контроллера в Spring MVC. Как и любой другой сервлет, DispatcherServlet должен быть настроен в файле web.xml веб-приложения. Поэтому первое, что необходимо сделать, – это поместить следующий элемент <servlet> в файл web.xml:

```
<servlet>
    <servlet-name>spitter</servlet-name>
    <servlet-class>
```

```
org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>
```

Имя сервлета в элементе `<servlet-name>` имеет большое значение. По умолчанию после загрузки сервлета `DispatcherServlet` он загрузит контекст приложения Spring из XML-файла с именем, соответствующим имени сервлета. В данном случае, поскольку сервлет получил имя `spitter`, `DispatcherServlet` будет пытаться загрузить контекст приложения из файла с именем `spitter-servlet.xml` (находящегося в каталоге `WEB-INF` приложения).

Далее необходимо указать, какие адреса URL будут обрабатываться сервлетом `DispatcherServlet`. Обычно сервлету `DispatcherServlet` передаются шаблоны адресов URL, такие как `*.htm`, `/*` или `/app`. Но эти шаблоны имеют следующие проблемы.

- ❑ Шаблон `*.htm` предполагает, что ответ всегда будет иметь формат HTML (что, как будет показано в главе 11, не всегда соответствует истинному положению дел).
- ❑ Шаблон `/*` не предполагает какого-то определенного формата ответа, но указывает, что `DispatcherServlet` будет обслуживать *все* запросы. Это усложняет обслуживание статических объектов, таких как изображения и таблицы стилей.
- ❑ Шаблон `/app` (или подобный ему) помогает отделить содержимое, обслуживаемое сервлетом `DispatcherServlet`, от остального содержимого. Но тогда нам придется реализовать обработку характерных особенностей наших URL (в частности, путь `/app`). Это ведет к сложному преобразованию адресов URL с целью скрыть путь `/app`.

Вместо того чтобы использовать какие-либо из этих проблематичных схем, я предпочитаю настраивать отображение `DispatcherServlet`, как показано ниже:

```
<servlet-mapping>
  <servlet-name>spitter</servlet-name>
  <url-pattern>/</url-pattern>
</servlet-mapping>
```

Отображая сервлет `DispatcherServlet` в адрес `/`, я сообщаю, что он является сервлетом по умолчанию, отвечающим за обработку всех запросов, включая запросы на получение статического содержимого.

Если вас беспокоит, как DispatcherServlet будет обрабатывать эти типы запросов, тогда задержитесь ненадолго. Небольшой трюк с настройками освободит вас как разработчика от необходимости заботиться о деталях. Пространство имен mvc в Spring включает новый элемент <mvc:resources>, автоматически обслуживающий запросы на получение статического содержимого. Все, что от вас требуется, – всего лишь определить его в конфигурации Spring.

Это означает, что пришло время создать файл spitter-servlet.xml, который сервер DispatcherServlet будет использовать для создания контекста приложения. В листинге 8.1 приводится начало файла spitter-servlet.xml.

Листинг 8.1. Элемент <mvc:resources> определяет порядок обработки статических ресурсов

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc
        http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <mvc:resources mapping="/resources/**"
        location="/resources/" />      
    </beans>                                <!-- статических ресурсов --&gt;</pre>
```

Как уже говорилось выше, все запросы, проходящие через DispatcherServlet, должны каким-то образом обрабатываться, часто с помощью других контроллеров. Поскольку запросы на получение статического содержимого также обрабатываются сервером DispatcherServlet, необходимо каким-то образом сообщить ему, как обрабатывать эти ресурсы. Но создание отдельного контроллера для этой цели выглядит слишком трудоемкой задачей. К счастью, эту работу может выполнить элемент <mvc:resources>¹.

Элемент <mvc:resources> определяет обработчик статического содержимого. Атрибуту mapping в примере присвоено значение /resources/**, включающее шаблонный символ в стиле утилиты Ant,

¹ Элемент <mvc:resources> появился в версии Spring 3.0.4. Если вы пользуетесь более старой версией Spring, он будет недоступен.

который указывает, что путь должен начинаться с `/resources` и может включать дополнительные элементы пути. Атрибут `location` определяет местоположение обслуживаемых файлов. Из данной конфигурации следует, что любые запросы, путь в которых начинается с `/resources`, автоматически будут адресоваться к папке `/resources`, находящейся в корневом каталоге приложения. То есть все изображения, таблицы стилей, сценарии JavaScript и другие статические ресурсы должны храниться в папке `/resources` приложения.

Теперь, когда мы разобрались с проблемой обработки статического содержимого, можно подумать о функциональности приложения. Поскольку мы находимся лишь в начале пути, начнем с создания простой домашней (или главной) страницы приложения Spitter.

8.2. Создание простого контроллера

Для обеспечения веб-функциональности приложения Spitter мы создадим контроллеры ресурсов. Но, вместо того чтобы писать отдельные контроллеры для каждого случая, мы создадим единственный контроллер, обрабатывающий все типы ресурсов, обслуживающих приложением.

Будучи простым приложением, приложение Spitter обладает ресурсами двух основных типов: пользователи приложения и сообщения, которыми пользователи обмениваются между собой. То есть необходимо написать контроллер, обслуживающий пользователей, и контроллер, обслуживающий сообщения. На рис. 8.2 показано, как эти контроллеры укладываются в общую схему приложения.

Помимо контроллеров, каждый из которых обслуживает свой ресурс, в приложении также потребуются два вспомогательных контроллера, как показано на рис. 8.2. Эти контроллеры обслуживаются дополнительные запросы, которые не отображаются на какой-то конкретный ресурс.

Один из этих контроллеров, `HomeController`, отвечает за отображение главной страницы приложения – страницы, которая не связана ни с пользователями, ни с сообщениями. Это будет первый контроллер, который мы напишем. Но сначала, поскольку мы разрабатываем контроллеры, управляемые аннотациями, выполним дополнительные настройки.

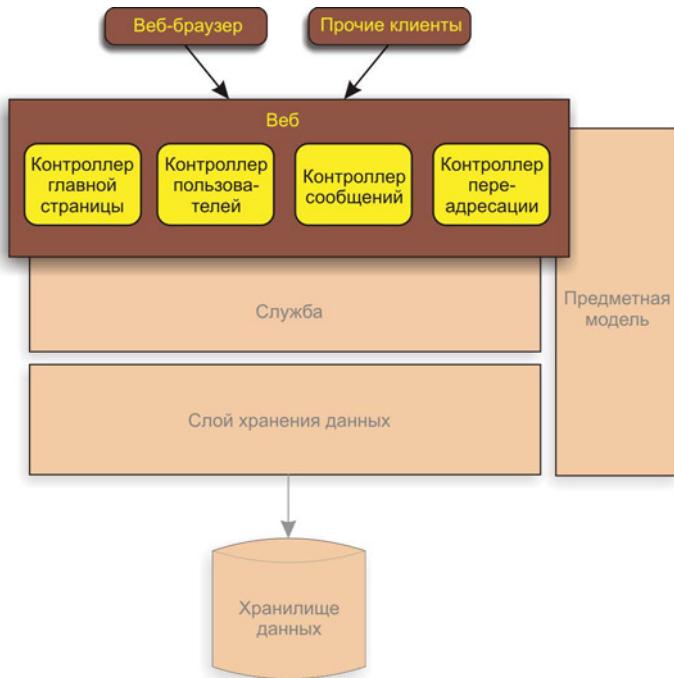


Рис. 8.2. Веб-слой приложения Spittle включает два контроллера ресурсов и два вспомогательных контроллера

8.2.1. Настройка поддержки аннотаций в Spring MVC

Как упоминалось выше, сервлет DispatcherServlet обращается к одному или нескольким механизмам отображения с целью определить, какому контроллеру передать запрос для обработки. В состав Spring входят несколько реализаций механизмов отображения, включая перечисленные ниже.

- ❑ BeanNameUrlHandlerMapping – отображает контроллеры на адреса URL, опираясь на имена компонентов контроллеров.
- ❑ ControllerBeanNameHandlerMapping – подобно BeanNameUrlHandlerMapping, отображает контроллеры на адреса URL, также опираясь на имена компонентов контроллеров. Но в данном случае не требуется, чтобы имена компонентов подбирались в соответствии с соглашениями об адресах URL.

- ❑ ControllerClassNameHandlerMapping – отображает контроллеры на адреса URL, используя имена классов контроллеров.
- ❑ DefaultAnnotationHandlerMapping – отображает запросы на контроллеры и методы контроллеров, отмеченные аннотацией @RequestMapping.
- ❑ SimpleUrlHandlerMapping – отображает контроллеры на адреса URL, используя свойство-коллекцию, объявленное в контексте приложения Spring.

Использование любого из этих механизмов отображения обычно сводится к его настройке как компонента Spring. Но если требуемый компонент механизма отображения не будет найден, тогда сервер DispatcherServlet автоматически создаст и будет использовать BeanNameUrlHandlerMapping и DefaultAnnotationHandlerMapping. К счастью, нас в первую очередь интересуют аннотированные классы контроллеров, поэтому реализация DefaultAnnotationHandlerMapping, используемая сервером DispatcherServlet по умолчанию, подходит как нельзя лучше.

DefaultAnnotationHandlerMapping отображает запросы на методы контроллеров, отмеченные аннотацией @RequestMapping (с которой мы встретимся в следующем разделе). Но в Spring MVC аннотации могут применяться не только для отображения запросов на методы. В процессе создания контроллеров мы также будем использовать аннотации для связывания параметров запросов с параметрами методов-обработчиков, выполнения проверки и преобразования сообщений. Поэтому одного компонента DefaultAnnotationHandlerMapping будет мало.

К счастью, достаточно добавить в файл spitter-servlet.xml всего одну строку, чтобы обрести возможность использовать все остальные аннотации, имеющиеся в Spring MVC:

```
<mvc:annotation-driven/>
```

Маленький элемент `<mvc:annotation-driven>` обладает большой властью. Он привносит некоторые дополнительные возможности, включая поддержку аннотаций JSR-303 для проверки данных, преобразования сообщений и форматирования полей.

Подробнее об этих возможностях будет рассказываться по мере необходимости. А сейчас создадим контроллер главной страницы.

8.2.2. Контроллер главной страницы

Обычно самое первое, что видят посетители веб-сайта, – это главная страница. Это парадный вход, обеспечивающий доступ ко всей

функциональности сайта. В случае с приложением Spitter основная задача главной страницы состоит в том, чтобы поприветствовать посетителя и отобразить перед ним несколько последних сообщений в надежде вызвать у него желание присоединиться к обсуждению.

В листинге 8.2 представлен контроллер HomeController – простейший контроллер Spring MVC, обслуживающий запросы на получение главной страницы.

Листинг 8.2. Контроллер HomeController приветствует пользователей приложения Spitter

```
package com.habuma.spitter.mvc;
import javax.inject.Inject;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spitter.service.SpitterService;

@Controller // Объявить как контроллер
public class HomeController {
    public static final int DEFAULT_SPITTLES_PER_PAGE = 25;

    private SpitterService spitterService;

    @Inject // Внедрить SpitterService
    public HomeController(SpitterService spitterService) {
        this.spitterService = spitterService;
    }

    @RequestMapping({"/", "/home"}) // Обрабатывать запросы на получение
                                    // главной страницы
    public String showHomePage(Map<String, Object> model) {

        model.put("spittles", spitterService.getRecentSpittles(
            DEFAULT_SPITTLES_PER_PAGE)); //Добавить сообщения в модель

        return "home"; // Вернуть имя представления
    }
}
```

Несмотря на свою простоту реализации контроллера HomeController, о ней многое можно сказать. Во-первых, аннотация @Controller указывает, что этот класс является классом контроллера. Данная анно-

тация является специализированной версией аннотации `@Component`, которая помогает элементу `<context:component-scan>` отыскивать и регистрировать классы с аннотацией `@Controller` как компоненты, как если бы они были отмечены аннотацией `@Component`.

Это означает, что в файле `spitterservlet.xml` необходимо настроить элемент `<context:component-scan>`, чтобы обеспечить автоматическое обнаружение и регистрацию класса `HomeController` (и всех других классов контроллеров, которые нам предстоит написать) как компонента. Ниже приводится соответствующий фрагмент в XML-файле:

```
<context:component-scan base-package="com.habuma.spitter.mvc" />
```

Вернемся к классу `HomeController`. Мы знаем, что ему потребуется извлекать список последних сообщений посредством компонента `SpitterService`. Поэтому был создан конструктор, принимающий `SpitterService` в виде аргумента и отмеченный аннотацией `@Inject`, которая обеспечивает автоматическое внедрение при создании экземпляра контроллера.

Фактическую работу контроллера выполняет метод `showHomePage()`. Как видно из листинга 8.2, он отмечен аннотацией `@RequestMapping`. Эта аннотация преследует две цели. Во-первых, она превращает метод `showHomePage()` в обработчик запросов. И, во-вторых, указывает, что этот метод должен вызываться для обработки запросов, содержащих путь `/` или `/home`.

Как обработчик запросов метод `showHomePage()` принимает отображение (`Map`) строк в объекты. Это отображение представляет модель – данные, передаваемые между контроллером и представлением. После получения списка последних сообщений (объектов `Spittle`) вызовом метода `getRecentSpittles()` компонента `SpitterService` этот список помещается в модель `model`, благодаря чему представление получит возможность отобразить его.

При создании других контроллеров будет показано, что сигнатура метода обработки запросов может включать практически любые аргументы. Даже при том, что методу `showHomePage()` необходимо только модель, ему можно было бы передать `HttpServletRequest`, `HttpServletResponse`, строки или числа, соответствующие параметрам запроса, значения в cookie, значения заголовков HTTP-запроса и массу других. Однако пока достаточно передать одну модель.

В конце метод `showHomePage()` должен вернуть строку с логическим именем представления, отображающего результаты. Класс контрол-

лера не должен принимать прямого участия в отображении результатов, он должен всего лишь определить реализацию представления, которое отобразит данные. Когда контроллер завершит свою работу, по указанному имени DispatcherServlet определит фактическую реализацию представления, обратившись к арбитру представлений.

Настройка арбитра представлений будет выполнена ниже, но сначала напишем модульный тест, чтобы убедиться, что HomeController действует, как это ожидается.

Тестирование контроллера

Самое примечательное в контроллере HomeController (как и в большинстве контроллеров Spring MVC) заключается в том, что он практически не содержит ничего, характерного для фреймворка Spring. Фактически, если отбросить три аннотации, получится обычный POJO.

С точки зрения модульного тестирования это очень важное обстоятельство, потому что оно означает, что класс HomeController легко можно протестировать без привлечения фиктивных или других объектов, зависящих от фреймворка Spring. В листинге 8.3 демонстрируется класс HomeControllerTest, реализующий тестирование контроллера HomeController.

Листинг 8.3. Тест, позволяющий убедиться в корректной работе контроллера HomeController

```
package com.habuma.spitter.mvc;

import static com.habuma.spitter.mvc.HomeController.*;
import static java.util.Arrays.*;
import static org.junit.Assert.*;
import static org.mockito.Mockito.*;

import java.util.HashMap;
import java.util.List;

import org.junit.Test;

import com.habuma.spitter.domain.Spittle;
import com.habuma.spitter.service.SpitterService;

public class HomeControllerTest {
    @Test
    public void shouldDisplayRecentSpittles() {
```

```
List<Spittle> expectedSpittles =  
    asList(new Spittle(), new Spittle(), new Spittle());  
  
    // Фиктивный объект SpitterService  
    SpitterService spitterService = mock(SpitterService.class);  
  
    when(spitterService.getRecentSpittles(DEFAULT_SPITTLES_PER_PAGE)).  
        thenReturn(expectedSpittles);  
  
    // Создать контроллер  
    HomeController controller =  
        new HomeController(spitterService);  
  
    HashMap<String, Object> model = new HashMap<String, Object>();  
  
    // Вызвать обработчик  
    String viewName = controller.showHomePage(model);  
  
    assertEquals("home", viewName);  
  
    // Проверить результаты  
    assertSame(expectedSpittles, model.get("spittles"));  
    verify(spitterService).getRecentSpittles(DEFAULT_SPITTLES_PER_PAGE);  
}  
}
```

Единственное, что необходимо контроллеру `HomeController` для выполнения своей работы, – экземпляр `SpitterService`, вместо которого передается фиктивная реализация, созданная с помощью фреймворка Mockito¹. После создания фиктивной реализации `SpitterService` остается лишь создать новый экземпляр класса `HomeController` и затем вызвать метод `showHomePage()`. В заключение проверяется, что список сообщений, возвращаемый фиктивной реализацией `SpitterService`, оказывается в модели с ключом `spittles` и что метод возвращает логическое имя `home` представления.

Как видите, тестирование контроллера Spring MVC мало чем отличается от тестирования любого другого POJO в приложении на основе Spring. Даже при том, что в конечном итоге контроллер будет использоваться для обслуживания веб-страницы, нам не пришлось делать что-то особенное для его тестирования.

¹ <http://mockito.org>.

На данный момент у нас имеется контроллер, обрабатывающий запросы на получение главной страницы, и тест, позволяющий убедиться, что контроллер выполняет свою работу. Но пока без ответа остается еще один вопрос. Метод `showHomePage()` возвращает логическое имя представления. Но как это имя будет использоваться для определения представления, отображающего страницу?

8.2.3. Поиск представлений

Последнее, что осталось сделать в ходе обработки запроса, – отобразить страницу. Для решения подобных задач используются представления – обычно JavaServer Pages (JSP), но могут использоваться и другие технологии реализации представлений, такие как Velocity и FreeMarker. Чтобы определить, какое представление должно обрабатывать данный запрос, `DispatcherServlet` обращается за помощью к арбитру представлений с целью заменить логическое имя, возвращаемое контроллером, ссылкой на фактическое представление, реализующее отображение результатов.

В действительности работа *арбитра представлений* заключается в отображении логического имени на некоторую реализацию интерфейса `org.springframework.web.servlet.View`. Но пока достаточно представлять себе арбитр представлений как некоторый механизм отображения имени представления в JSP, что он и делает.

В состав Spring входят несколько реализаций арбитров представлений, которые перечислены в табл. 8.1.

Таблица 8.1. Когда дело доходит до отображения информации, фреймворк Spring MVC позволяет выбрать соответствующее представление с помощью одной из следующих реализаций арбитров представлений

Арбитр представлений	Описание
BeanNameViewResolver	Отыскивает реализацию интерфейса <code>View</code> , объявленную с помощью элемента <code><bean></code> с идентификатором, совпадающим с логическим именем представления
ContentNegotiatingViewResolver	Для определения представления использует один или более других арбитров представлений, выбирая нужного арбитра, исходя из типа запрошенного содержимого. (Подробнее об этом арбитре представлений рассказывается в главе 11)

Таблица 8.1 (продолжение)

Арбитр представлений	Описание
FreeMarkerViewResolver	Отыскивает шаблон FreeMarker, путь к которому определяется за счет добавления приставки и окончания к логическому имени представления
InternalResourceViewResolver	Отыскивает шаблон представления, содержащийся в WAR-файле веб-приложения. Путь к шаблону определяется добавлением приставки и окончания к логическому имени представления
JasperReportsViewResolver	Отыскивает шаблон представления, объявленный как отчет Jasper Reports, путь к которому определяется за счет добавления приставки и окончания к логическому имени представления
ResourceBundleViewResolver	Отыскивает реализацию интерфейса View в файле свойств
TilesViewResolver	Отыскивает представление, которое определено как шаблон Tiles, путь к которому определяется за счет добавления приставки и окончания к логическому имени представления
UrlBasedViewResolver	Это базовый класс для реализации некоторых других арбитров представлений, таких как InternalResourceViewResolver. Может использоваться как самостоятельный арбитр, но он не обладает такой широтой возможностей, как его подклассы. Например, UrlBasedViewResolver не способен выявлять представления, опираясь на текущие региональные настройки
VelocityLayoutViewResolver	Подкласс класса VelocityViewResolver, который поддерживает компоновку страниц посредством VelocityLayoutView (реализация представления, имитирующего VelocityLayoutServlet)
VelocityViewResolver	Арбитр представлений, созданных на основе фреймворка Velocity, где путь шаблона Velocity определяется за счет добавления приставки и окончания к логическому имени представления
XmlViewResolver	Отыскивает реализацию интерфейса View, объявленную с помощью элемента <bean> в XML-файле (/WEB-INF/views.xml). Этот арбитр представлений близко напоминает BeanNameViewResolver, за исключением того, что элемент <bean> представления определяется отдельно от других компонентов контекста приложения Spring



Таблица 8.1 (окончание)

Арбитр представлений	Описание
XsltViewResolver	Отыскивает представления, реализованные на основе XSLT-определения, где путь к таблице стилей XSLT определяется за счет добавления приставки и окончания к логическому имени представления

У нас недостаточно ни времени, ни места, чтобы познакомиться с каждым из этих арбитров представлений. Но некоторые из них стоят того, чтобы взглянуть на них поближе. Начнем с арбитра `InternalResolverViewResolver`.

Поиск внутренних представлений

Фреймворк Spring MVC во многом следует принципу *преимущества соглашений перед настройками*. Арбитр InternalResourceViewResolver является одним из таких элементов, ориентированных на соглашения. Он отображает логическое имя представления в объект View, который перекладывает ответственность за отображение страницы на шаблон (обычно JSP), находящийся в контексте веб-приложения. Как показано на рис. 8.3, он выполняет такое отображение, добавляя приставку и окончание к логическому имени представления, в результате получая путь к шаблону, являющемуся ресурсом внутри веб-приложения.



Рис. 8.3. InternalResourceViewResolver определяет путь к шаблону представления, добавляя указанные приставку и окончание к логическому имени представления

Допустим, что все страницы JSP для приложения Spitter находятся в каталоге `/WEB-INF/views/`. Учитывая это условие, компонент `InternalResourceViewResolver` необходимо настроить в файле `spitter-servlet.xml`, как показано ниже:

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

Когда DispatcherServlet запросит у арбитра InternalResourceViewResolver представление, он добавит к логическому имени приставку /WEB-INF/views/ и окончание .jsp. В результате получится путь к странице JSP для отображения вывода. Затем InternalResourceViewResolver передаст этот путь объекту View, который направит запрос странице JSP. То есть, когда HomeController вернет логическое имя представления home, в конечном итоге будет получен путь /WEB-INF/views/home.jsp.

По умолчанию объект View, созданный арбитром InternalResourceViewResolver, является экземпляром класса InternalResourceView, который просто передает запрос странице JSP для отображения. Но, поскольку home.jsp использует некоторые теги JSTL, объект InternalResourceView можно заменить объектом JstlView, определив свойство viewClass, как показано ниже:

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

Представление JstlView направляет запрос странице JSP, как и InternalResourceView. Но дополнительно устанавливает JSTL-атрибуты запроса, что позволяет пользоваться преимуществом наличия поддержки интернационализации в JSTL.

Мы не будем погружаться в изучение FreeMarkerViewResolver, JasperReportsViewResolver, VelocityViewResolver, VelocityLayoutViewResolver и XsltViewResolver, однако отмечу, что в поисках шаблона представления они, подобно арбитру InternalResourceViewResolver, определяют представления, добавляя приставку и окончание к логическому имени представления. Теперь, после знакомства с принципом действия арбитра InternalResourceViewResolver, особенности работы других арбитров представлений должны выглядеть более естественными.

Арбитр `InternalResourceViewResolver`, отыскивающий JSP-представления, отлично подходит для использования в простых веб-приложениях, отображающих несложные страницы. Но веб-сайты часто реализуют весьма интересные пользовательские интерфейсы, используя элементы, общие для разных страниц. Для реализации таких веб-сайтов часто используются диспетчеры компоновки, такие как Apache Tiles. Посмотрим, как настроить Spring MVC на использование представлений, входящих в состав фреймворка Tiles.

Поиск представлений Tiles

Apache Tiles¹ – это фреймворк обслуживания шаблонов для соединения полных страниц из фрагментов во время выполнения. Первоначально он создавался как часть фреймворка Struts, но теперь фреймворк Tiles используется многими другими веб-фреймворками. Мы будем использовать его совместно с фреймворком Spring MVC, чтобы обеспечить компоновку страниц приложения Spitter.

Чтобы использовать представления из фреймворка Tiles в Spring MVC, сначала нужно зарегистрировать `TilesViewResolver` как компонент приложения Spring в файле `spitter-servlet.xml`:

```
<bean class="org.springframework.web.servlet.view.tiles2.TilesViewResolver"/>
```

Этот скромный элемент `<bean>` настраивает арбитр представлений, который будет искать представления, являющиеся определениями шаблонов Tiles, имена которых совпадают с логическими именами представлений.

В этом определении отсутствует информация об определениях шаблонов Tiles. Сам по себе арбитр `TilesViewResolver` ничего не знает о шаблонах Tiles и в попытках отыскать их опирается на компонент `TilesConfigurer`, хранящий нужную информацию. Поэтому добавим в файл `spitter-servlet.xml` определение компонента `TilesConfigurer`:

```
<bean class="org.springframework.web.servlet.view.tiles2.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/views/**/views.xml</value>
        </list>
    </property>
</bean>
```

¹ <http://tiles.apache.org>.

Компонент `TilesConfigurer` загружает один или более файлов шаблонов `Tiles` и делает их доступными для `TilesViewResolver`. Нам потребуется несколько файлов с определениями шаблонов `Tiles` для приложения `Spitter`. Все файлы будут иметь имя `views.xml` и храниться в разных подкаталогах в каталоге `/WEB-INF/views`. Поэтому в определении выше свойству `definitions` присвоено значение `/WEB-INF/views/**/views.xml`. Шаблонный символ `**` в стиле утилиты `Ant` указывает, что поиск файлов `views.xml` должен выполняться по всему дереву каталогов с корнем в каталоге `/WEB-INF/views`.

Что касается самих файлов `views.xml`, мы будем создавать их на протяжении всей главы, начав с файла, определяющего шаблон для главной страницы. В листинге 8.4 представлен файл `views.xml`, определяющий шаблон главной страницы, а также общий шаблон, который будет использоваться другими шаблонами.

Листинг 8.4. Определение шаблона `Tiles`

```
<!DOCTYPE tiles-definitions PUBLIC
  "-//Apache Software Foundation//DTD Tiles Configuration 2.1//EN"
  "http://tiles.apache.org/dtds/tiles-config_2_1.dtd">

<tiles-definitions>
  <definition name="template"
    template="/WEB-INF/views/main_template.jsp" <!-- Общий шаблон -->
    <put-attribute name="top"
      value="/WEB-INF/views/tiles/spittleForm.jsp" />
    <put-attribute name="side"
      value="/WEB-INF/views/tiles/signinSignup.jsp" />
  </definition>

  <definition name="home" extends="template"> <!-- Шаблон главной страницы -->
    <put-attribute name="content" value="/WEB-INF/views/home.jsp" />
  </definition>
</tiles-definitions>
```

Определение `home` наследует определение `template` и для отображения основного содержимого использует JSP-страницу `home.jsp`, но для предоставления дополнительной информации опирается на шаблон `template`.

Именно этот шаблон `home` будет отыскивать арбитр `TilesViewResolver` при преобразовании логического имени представления, возвращающего методом `showHomePage()` класса `HomeController`. Для отображения

результатов с применением шаблона `home` контроллер `DispatcherServlet` будет передавать запросы фреймворку `Tiles`.

8.2.4. Объявление представления главной страницы

Как видно в листинге 8.4, главная страница конструируется из нескольких отдельных фрагментов. Файл `main_template.jsp` описывает общую структуру всех страниц в приложении `Spitter`, тогда как `home.jsp` отображает лишь основное содержимое главной страницы. Плюс некоторые общие элементы, объявленные в файлах `spittleForm.jsp` и `signinsignup.jsp`.

Теперь сосредоточимся на файле `home.jsp` как наиболее уместном при обсуждении главной страницы. Именно в этой странице JSP завершается путешествие запроса. Она принимает список сообщений, которые контроллер `HomeController` поместил в модель, и отображает их в веб-странице. В листинге 8.5 представлено содержимое файла `home.jsp`.

Листинг 8.5. Элемент `<div>` главной страницы, вставляемый в шаблон

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="s" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="t" uri="http://tiles.apache.org/tags-tiles"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>

<div>
    <h2>A global community of friends and strangers spitting out their
    inner-most and personal thoughts on the web for everyone else to
    see.</h2>
    <h3>Look at what these people are spitting right now...</h3>

    <ol class="spittle-list">
        <c:forEach var="spittle" items="${spittles}"> <!-- Цикл по списку сообщений -->

            <s:url value="/spitters/{spitterName}"
                   var="spitter_url" > <!-- Конструирование URL-сообщения -->
            <s:param name="spitterName"
                     value="${spittle.spitter.username}" />
        </s:url>

        <li>
            <span class="spittleListImage">
```

```
<img src=
    "http://s3.amazonaws.com/spitterImages/${spittle.spitter.id}.jpg"
    width="48"
    border="0"
    align="middle"
    onError=
    "this.src='<s:url value="/resources/images"/>/spitter_avatar.png';"/>
</span>
<span class="spittleListText">
    <a href="${spitter_url}">    <!-- Отображение свойств сообщения --&gt;
        &lt;c:out value="${spittle.spitter.username}" /&gt;&lt;/a&gt;
    - &lt;c:out value="${spittle.text}" /&gt;&lt;br/&gt;
    &lt;small&gt;&lt;fmt:formatDate value="${spittle.when}"
        pattern="hh:mma MMM d, yyyy" /&gt;&lt;/small&gt;
    &lt;/span&gt;
&lt;/li&gt;
&lt;/c:forEach&gt;
&lt;/ol&gt;
&lt;/div&gt;</pre>
```

Основная работа в файле `home.jsp`, помимо вывода нескольких приветственных сообщений в начале, выполняется в теге `<c:forEach>`, который в цикле обходит список сообщений и отображает информацию о каждом из них. Поскольку сообщения (объекты `Spittle`) находятся внутри модели с ключом `spittles`, ссылка на список может быть получена с использованием конструкции `${spittles}`.

Атрибуты модели и запроса: взгляд изнутри. В примере выше этого не видно, но конструкция `${spittles}` в файле `home.jsp` ссылается на сервер-лет атрибута запроса с именем `spittles`. После того, как `HomeController` выполнит свою работу, и перед тем, как управление будет передано странице `home.jsp`, контроллер `DispatcherServlet` скопирует все элементы модели в атрибут запроса с тем же именем.

Обратите внимание на тег `<s:url>` в середине файла. Здесь этот тег используется для создания сервлета относительного URL-сообщения в приложении `Spitter`. Тег `<s:url>` появился в версии Spring 3.0 и действует подобно JSTL-тегу `<c:url>`.

Главное отличие между тегами `<s:url>` и `<c:url>` состоит в том, что `<s:url>` поддерживает параметризованные пути в адресах URL. В данном случае путь параметризуется именем пользователя приложения `Spitter`. Например, если предположить, что пользователь при-

ложении Spitter имеет имя *habuma*, а сервлет контекста имеет имя *Spitter*, тогда в результате получится путь /Spitter/spitters/habuma.

При отображении эта страница JSP будет скомпонована с другими страницами JSP, объявленными в том же определении Tiles, в результате получится главная страница приложения Spitter, как показано на рис. 8.4.



Рис. 8.4. Главная страница приложения Spitter с приветствием и списком последних сообщений

К настоящему моменту мы создали первый контроллер Spring MVC, настроили арбитр представлений и определили простое представление JSP для отображения результатов обращения к контроллеру. Но осталась одна маленькая проблема. В контроллере HomeController нас подкарауливает исключение, потому что DispatcherServlet понятия не имеет, где искать компонент SpitterService. К счастью, это легко исправить.

8.2.5. Завершение определения контекста приложения Spring

Как уже упоминалось выше, DispatcherServlet загружает контекст приложения Spring из единственного XML-файла, имя которого ос-

новано на имени, указанном в элементе <servlet-name>. Но как быть с остальными компонентами, объявленными в предыдущих главах, такими как SpitterService? Если DispatcherServlet загружает компоненты из файла с именем spitter-servlet.xml, разве мы не должны объявить остальные компоненты в этом же файле?

В предыдущих главах конфигурация Spring была разбита на несколько XML-файлов: один – для слоя службы, один – для слоя хранения данных и один – для настройки источника данных. Хотя это и не обязательно, тем не менее многие предпочитают распределить конфигурацию Spring по нескольким файлам. Учитывая это, кажется вполне оправданным поместить все настройки веб-слоя в файл spitter-servlet.xml, загружаемый контроллером DispatcherServlet. Но нам необходим некоторый способ загрузить все остальные конфигурационные файлы.

В этом нам поможет ContextLoaderListener – сервлет, загружающий дополнительные конфигурационные файлы в контекст приложения Spring, созданный контроллером DispatcherServlet. Чтобы задействовать ContextLoaderListener, нужно добавить следующий элемент <listener> в файл web.xml:

```
<listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
</listener>
```

Кроме того, сервлету ContextLoaderListener необходимо сообщить, какие конфигурационные файлы Spring он должен загрузить. Если не указано конкретное имя файла, он будет искать файл /WEB-INF/applicationContext.xml. Но в случае, когда конфигурация разбита на несколько файлов, возможности загружать единственный файл явно недостаточно, поэтому следует переопределить поведение по умолчанию.

Чтобы сообщить сервлету ContextLoaderListener имена нескольких конфигурационных файлов Spring, необходимо определить параметр contextConfigLocation в контексте сервлета:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/spitter-security.xml
```

```
classpath:service-context.xml  
classpath:persistence-context.xml  
classpath:dataSource-context.xml  
</param-value>  
</context-param>
```

Параметр `contextConfigLocation` определяется как список путей к файлам. Если явно не указано иное, пути к файлам интерпретируются относительно корневого каталога приложения. Но, так как конфигурация Spring в данном случае разбита на множество файлов, разбросанных по нескольким JAR-файлам в веб-приложении, в примере выше к некоторым из них был добавлен префикс `classpath:`, чтобы обеспечить возможность их загрузки из библиотеки классов приложения.

Выше можно заметить, что в настройки включены все конфигурационные файлы Spring, созданные в предыдущих главах. Обратите также внимание на несколько дополнительных конфигурационных файлов, которые еще не рассматривались, – они будут обсуждаться в последующих главах.

Теперь, после создания первого контроллера, приложение Spitter готово обрабатывать запросы на получение главной страницы. Если бы для работы приложения достаточно было одной главной страницы, работу на этом можно было бы считать законченной. Но приложение Spitter должно иметь не только главную страницу, поэтому продолжим конструирование приложения. Следующим нашим шагом будет создание контроллера, обрабатывающего входные данные.

8.3. Контроллер обработки входных данных

Контроллер `HomeController` получился довольно простым. Перед ним не стоит задача обрабатывать пользовательские данные или какие-либо параметры. Он просто обрабатывает простейший запрос и заполняет модель данными для представления. Сложно было бы придумать более простой контроллер.

Но не у всех контроллеров жизнь такая же простая. Часто от контроллеров требуется выполнять операции с некоторыми данными, которые передаются в виде параметров запроса в URL или данных формы. Это как раз относится к контроллерам `SpitterController` и `SpittleController`. Эти два контроллера будут обрабатывать различные виды запросов, многие из которых содержат некоторые входные данные.

Примером одной из задач, которые должен решать контроллер SpitterController, является обработка входных данных, определяющих пользователя, с целью отобразить список его сообщений. Приступим к реализации данной функциональности и посмотрим, как писать контроллеры, обрабатывающие входные данные.

8.3.1. Создание контроллера, обрабатывающего входные данные

Один из способов передачи данных контроллеру SpitterController – отправка имени пользователя приложения Spitter в URL в виде параметра запроса. Например, при обращении по URL <http://localhost:8080/spitter/spitters/spittles?spitter=habuma> приложение должно отобразить все сообщения пользователя habuma.

В листинге 8.6 представлена реализация контроллера SpitterController, способного обрабатывать такого рода запросы.

Листинг 8.6. Типичный способ обработки запросов на получение сообщений пользователя приложения Spitter

```
package com.habuma.spitter.mvc;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.service.SpitterService;
import static org.springframework.web.bind.annotation.RequestMethod.*;

@Controller
@RequestMapping("/spitter") // Корневой путь в URL
public class SpitterController {

    private final SpitterService spitterService;

    @Inject
    public SpitterController(SpitterService spitterService) {
        this.spitterService = spitterService;
    }

    // Обрабатывает GET-запросы к URL /spitter/spittles
    @RequestMapping(value="/spittles", method=GET)
```

```
public String listSpittlesForSpitter()
    @RequestParam("spitter") String username, Model model) {
    Spitter spitter = spitterService.getSpitter(username);
    model.addAttribute(spitter); // Заполнение модели
    model.addAttribute(spitterService.getSpittlesForSpitter(username));
    return "spittles/list";
}
}
```

В листинге 8.6 класс `SpitterController` отмечен аннотациями `@Controller` и `@RequestMapping`. Как уже говорилось выше, аннотация `@Controller` подсказывает элементу `<context:component-scan>`, что данный класс должен автоматически регистрироваться в контексте приложения Spring как компонент.

Обратите внимание, что класс `SpitterController` также отмечен аннотацией `@RequestMapping`. Аннотация `@RequestMapping` уже использовалась в определении класса `HomeController` для аннотирования метода `showHomePage()`, но при применении на уровне класса аннотация `@RequestMapping` действует несколько иначе.

При применении к классу, как в данном примере, аннотация `@RequestMapping` определяет корневой путь в URL, обрабатываемый данным контроллером. В конечном итоге в классе `SpitterController` будет реализовано несколько методов-обработчиков, по одному для каждого типа запросов. И в данном случае аннотация `@RequestMapping` сообщает, что все эти запросы будут иметь пути, начинающиеся со `/spitters`.

В настоящий момент в классе `SpitterController` имеется всего один метод: `listSpittlesForSpitter()`. Подобно любому другому методу-обработчику, данный метод отмечен аннотацией `@RequestMapping`. Она мало чем отличается от аналогичной аннотации, использованной в классе `HomeController`. Но в действительности разница гораздо существеннее, чем кажется на первый взгляд.

Действительно ли нужна аннотация `@RequestParam`? Аннотация `@RequestParam` не является строго обязательной. Ее удобно использовать для связывания параметров запроса с параметрами метода, когда их имена отличаются. В соответствии с соглашениями все неаннотированные параметры метода обработчика будут связаны с одноименными параметрами запроса. Например, метод `listSpittlesForSpitter()` имеет параметр `spitter`, которому соответствует параметр запроса `username`, поэтому аннотация `@RequestParam` в данном случае необходима.

Аннотация `@RequestParam` также может пригодиться, когда предполагается компилировать программный код Java без включения отладочной инфор-

мации. В этом случае имя параметра метода будет утрачено, и заодно будет утрачена возможность автоматического связывания параметров запроса с параметрами метода. По этой причине лучше всегда использовать аннотацию `@RequestParam` и не полагаться на имеющиеся соглашения.

Аннотация `@RequestMappings` ограничивает область отображения, объявленную в аннотации `@RequestMapping` на уровне класса. В данном случае на уровне класса класс `SpitterController` отображается на путь `/spitters`, а на уровне метода – на путь `/spittles`. В результате объединения этих двух аннотаций получается, что метод `listSpittlesForSpitter()` должен обрабатывать запросы для `/spitters/spittles`. Кроме того, значение `GET` в атрибуте `method` указывает, что этот метод будет обрабатывать только запросы типа HTTP `GET`.

Метод `listSpittlesForSpitter()` принимает строковый параметр `username` с именем пользователя и объект `Model`.

Параметр `username` отмечен аннотацией `@RequestParam("spitter")`, чтобы показать, что он должен получить значение параметра запроса с именем `spitter`. Этот параметр будет использоваться методом `listSpittlesForSpitter()` для поиска соответствующего объекта `Spitter` и его списка сообщений – объектов `Spittle`.

Возможно, вы уже призадумались над вторым параметром метода `listSpittlesForSpitter()`. В контроллере `HomeController` в качестве модели передавался объект типа `Map<String, Object>`. А здесь передается объект `Model`.

В действительности объект, передаваемый как параметр типа `Model`, можно передавать как параметр типа `Map<String, Object>`. Но тип `Model` позволяет использовать некоторые вспомогательные методы, удобные для заполнения модели, такие как `addAttribute()`. Метод `addAttribute()` действует практически так же, как метод `put()` интерфейса `Map`, за исключением того, что он определяет собственный ключ для доступа к данным в отображении.

Когда в модель добавляется объект `Spitter`, метод `addAttribute()` дает ему имя `spitter`, которое получается при применении правил именования свойств компонентов JavaBeans к имени класса объекта. При добавлении списка сообщений (объект `List`, содержащий список объектов `Spittle`) метод `addAttribute()` добавляет имя `List` к имени типа элементов списка, в результате получается атрибут с именем `spittleList`.

Теперь практически все готово к вызову метода `listSpittlesForSpitter()`. Мы определили контроллер `SpitterController` и его метод-

обработчик. Осталось лишь написать представление, которое будет отображать список сообщений.

8.3.2. Представление, отображающее список сообщений

Отображение списка сообщений в данном случае мало чем отличается от вывода списка сообщений на главной странице. Достаточно будет вывести имя пользователя (чтобы было ясно, кому принадлежат сообщения) и ниже перечислить все его сообщения.

Для этого прежде всего необходимо создать новое определение шаблона. Метод `listSpittlesForSpitter()` возвращает логическое имя представления `spittles/list`, а решить поставленную задачу нам поможет следующее определение шаблона:

```
<definition name="spittles/list" extends="template">
    <put-attribute name="content"
        value="/WEB-INF/views/spittles/list.jsp" />
</definition>
```

Подобно шаблону `home`, этот шаблон добавляет страницу JSP в атрибут `content` для отображения внутри `main_template.jsp`. В листинге 8.7 представлено содержимое файла `list.jsp`, используемого для отображения списка сообщений.

Листинг 8.7. Файл list.jsp – страница JSP, используемая для отображения списка сообщений

```
<%@ taglib prefix="s" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<div>
    <h2>Spittles for ${spitter.username}</h2>    <!-- Вывод имени пользователя -->

    <table cellspacing="15">
        <c:forEach items="${spittleList}"
            var="spittle">                                <!-- Вывод списка сообщений -->
            <tr>
                <td>
                    <img src=<s:url value="/resources/images/spitter_avatar.png"/></img>
                    width="48" height="48" /></td>
                <td>
                    <a href=<s:url value="/spitters/${spittle.spitter.username}"%>>
```

```
    ${spittle.spitter.username}</a>
    <c:out value="${spittle.text}" /><br/>
    <c:out value="${spittle.when}" />
  </td>
</tr>

</c:forEach>
</table>
</div>
```

Отбросив эстетическую сторону дела, эта страница JSP реализует все, что необходимо. В начале файла выводится заголовок с именем пользователя, которому принадлежат сообщения. Этот заголовок ссылается на свойство `username` объекта `Spitter`, который метод `listSpittlesForSpitter()` поместил в модель под именем `${spitter.username}`.

Самая интересная часть этой страницы JSP – итерации по списку сообщений и вывод информации о них. Атрибут `items` JSTL-тега `<c:forEach>` ссылается на список с именем `${spittleList}`, данным ему методом `addAttribute()` модели `Model`.

В данной реализации есть один маленький недостаток, на который следует обратить внимание, – здесь в качестве аватара пользователя используется жестко определенный файл `spitter_avatar.png`. В разделе 8.5 будет показано, как дать пользователю возможность выгружать собственные изображения для использования в качестве аватара.

Результат отображения содержимого списка сообщений страницы `list.jsp` представлен на рис. 8.5.

Но сначала необходимо дать пользователям возможность зарегистрироваться в приложении. Для этого необходимо написать контроллер, обрабатывающий данные, отправляемые с формой.

8.4. Обработка форм

При работе с формами в веб-приложении выполняются две операции: отображение формы и обработка данных, отправленных пользователем вместе с формой. Таким образом, чтобы зарегистрировать нового пользователя приложения `Spitter`, необходимо добавить в класс `SpitterController` два метода-обработчика для выполнения этих двух операций. Прежде чем форма с данными попадет на сервер, она должна быть отображена в браузере, поэтому

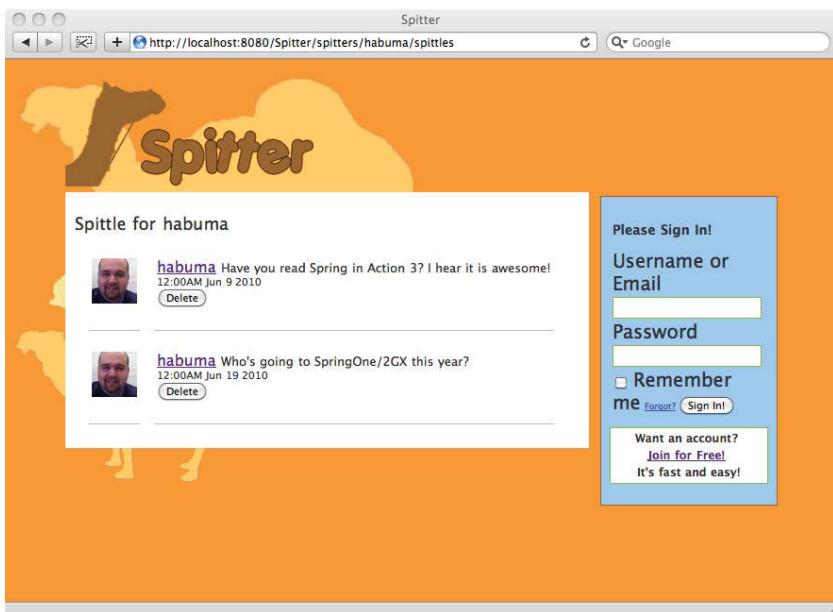


Рис. 8.5. При объединении с другими элементами шаблона Tiles страница list.jsp отображает список сообщений, принадлежащих указанному пользователю

начнем с метода-обработчика, реализующего отображение формы регистрации.

8.4.1. Отображение формы регистрации

Для отображения формы необходим объект Spitter, свойства которого будут связаны с полями формы. Поскольку форма предназначена для регистрации нового пользователя, достаточно будет создать новый, неинициализированный объект Spitter. В листинге 8.8 представлен метод-обработчик createSpitterProfile(), создающий объект Spitter и помещающий его в модель.

Листинг 8.8. Отображение формы регистрации пользователя

```
@RequestMapping(method=RequestMethod.GET, params="new")
public String createSpitterProfile(Model model) {
    model.addAttribute(new Spitter());
    return "spitters/edit";
}
```

Подобно другим методам-обработчикам, `createSpitterProfile()` отмечен аннотацией `@RequestMapping`. Но, в отличие от предыдущих методов-обработчиков, этот не требует определять путь в URL. То есть данный метод обрабатывает запросы для пути в URL, определяемом в аннотации `@RequestMapping` на уровне класса, в данном случае `/spitters`.

Единственное, что определяет аннотация `@RequestMapping` для этого метода, что он будет обрабатывать только запросы типа HTTP GET. Обратите также внимание на атрибут `params`, которому присваивается значение `new`. Это означает, что данный метод будет обрабатывать запросы HTTP GET для пути `/spitters` в URL, только если запрос включает параметр `new`. Эту разновидность URL, которые будут обрабатываться методом `createSpitterProfile`, иллюстрирует рис. 8.6.



Рис. 8.6. Атрибут `params` аннотации `@RequestMapping` может ограничивать область применения метода-обработчика запросами, содержащими определенные параметры

Для внутренних нужд метод `createSpitterProfile()` создает новый экземпляр `Spitter` и добавляет его в модель. По завершении он возвращает логическое имя представления `spitters/edit`, реализующего отображение формы.

Теперь создадим само представление.

Определение представления формы

Как и прежде, логическое имя представления, возвращаемое методом `createSpitterProfile()`, должно в конечном счете превратиться в определение шаблона Tiles формы для отображения в браузере. Поэтому добавим файл определения шаблона с именем `spitters/edit` в конфигурационный файл Tiles. Все необходимые определения со средоточены в следующем элементе `<definition>`.

```
<definition name="spitters/edit" extends="template">
    <put-attribute name="content"
        value="/WEB-INF/views/spitters/edit.jsp" />
</definition>
```

Как и ранее, атрибут `content` определяет, где находится основной шаблон страницы. В данном случае – в JSP-файле `/WEB-INF/views/spitters/edit.jsp`, содержимое которого представлено в листинге 8.9.

Листинг 8.9. Определение шаблона формы для ввода регистрационной информации

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>

<div>
<h2>Create a free Spitter account</h2>

<sf:form method="POST" modelAttribute="spitter">      <!-- Связать форму -->
    <fieldset>                                         <!-- с атрибутом модели -->
        <table cellspacing="0">
            <tr>
                <th><label for="user_full_name">Full name:</label></th>
                <td><sf:input path="fullName" size="15" id="user_full_name"/></td>
            </tr>
            <tr>
                <th><label for="user_screen_name">Username:</label></th>
                <td><sf:input path="username" size="15" maxlength="15"
                               id="user_screen_name"/>      <!-- Поле имени пользователя -->
                    <small id="username_msg">No spaces, please.</small>
                </td>
            </tr>
            <tr>
                <th><label for="user_password">Password:</label></th>
                <td><sf:password path="password" size="30"
                               showPassword="true"
                               id="user_password"/>          <!-- Поле пароля -->
                    <small>6 characters or more (be tricky!)</small>
                </td>
            </tr>
            <tr>
                <th><label for="user_email">Email Address:</label></th>
                <td><sf:input path="email" size="30"
                               id="user_email"/>           <!-- Поле электронной почты -->
                    <small>In case you forget something</small>
                </td>
            </tr>
            <tr>
                <th></th>
                <td>
                    <sf:checkbox path="updateByEmail"
                               id="user_send_email_newsletter"/> <!-- Признак необходимости -->
                </td>
            </tr>
        </table>
    </fieldset>
</sf:form>
```

```
<label for="user_send_email_newsletter"> <!-- присыпать уведомления -->
    Send me email updates!</label>           <!-- об изменениях -->
</td>
</tr>
</table>
</fieldset>
</sf:form>
</div>
```

Отличает этот JSP-файл от подобных ему, созданных до сих пор, использование библиотеки, входящей в состав Spring и применяемой для связывания форм. Тег `<sf:form>` связывает объект Spitter (определенный атрибутом `modelAttribute`), который метод `createSpitterProfile()` поместил в модель, с различными полями формы.

Теги `<sf:input>`, `<sf:password>` и `<sf:checkbox>` имеют атрибут `path`, ссылающийся на свойство объекта Spitter, связываемое с формой. Когда форма будет отправлена пользователем на сервер, значения полей будут сохранены в соответствующих свойствах объекта Spitter для дальнейшей обработки.

Обратите внимание: в теге `<sf:form>` определяется, что форма будет отправляться серверу в виде запроса HTTP POST. Но в ней не определяется адрес URL. Так как URL не указан, форма будет отправляться обратно по тому же пути в URL `/spitters`, который использовался для отображения формы. Это означает, что следующим шагом должно быть создание еще одного метода-обработчика, принимающего запросы POST для пути `/spitters`.

8.4.2. Обработка данных формы

Реализовав возможность отправки формы пользователем, мы должны создать метод-обработчик, принимающий объект Spitter (заполненный данными из формы) и сохраняющий его. И отображающий страницу с настройками пользователя. В листинге 8.10 представлен метод `addSpitterFromForm()`, обрабатывающий данные из формы.

Листинг 8.10. Метод addSpitter обрабатывает данные формы

```
@RequestMapping(method=RequestMethod.POST)
public String addSpitterFromForm(@Valid Spitter spitter,
                                  BindingResult bindingResult) {
    if(bindingResult.hasErrors()) {                                // Проверка ошибок
```

```
        return "spitters/edit";
    }
    spitterService.saveSpitter(spitter);           // Сохранить объект Spitter
    return "redirect:/spitters/" + spitter.getUsername(); // Переадресовать
}                                                 // после запроса POST
```

Обратите внимание, что метод `addSpitterFromForm()` отмечен аннотацией `@RequestMapping`, мало отличающейся от аннотации `@RequestMapping`, украшающей метод `createSpitterProfile()`. Ни одна из них не определяет путь в URL, поэтому оба метода будут обрабатывать запросы для пути `/spitters` в URL. Разница состоит в том, что `createSpitterProfile()` обрабатывает GET-запросы, а `addSpitterFromForm()` обрабатывает POST-запросы. Здесь все в порядке, потому что именно так отправляются формы.

После отправки формы ее поля в запросе будут связаны со свойствами объекта `Spitter`, который затем будет передан методу `addSpitterFromForm()`. Далее объект передается методу `saveSpitter()` компонента `SpitterService` для сохранения в базе данных.

В листинге 8.10 можно также заметить, что параметр `Spitter` отмечен аннотацией `@Valid`. Она указывает, что объект `Spitter` должен подвергаться проверке перед передачей методу. Подробнее об этой проверке рассказывается в следующем разделе.

Подобно методам-обработчикам, созданным ранее, этот метод также возвращает строку, определяющую, куда дальше должен направляться запрос. На этот раз вместо логического имени представления возвращается специальное представление переадресации. Префикс `redirect:` свидетельствует о том, что запрос должен быть переадресован в путь, указанный вслед за префиксом. Выполняя переадресацию на другую страницу, можно избежать двойной отправки формы, если пользователь щелкнет на кнопке **Refresh** (Обновить) в браузере.

Что касается пути переадресации – он примет вид `/spitters/{username}`, где `{username}` представляет только что полученное имя пользователя приложения `Spitter`. Например, если пользователь регистрируется под именем `habuma`, тогда после отправки формы он будет переадресован по адресу `/spitters/habuma`.

Обработка запросов, в которых путь определяется переменной

Теперь необходимо разобраться, как реализовать обработку запросов к пути `/spitters/{username}`. Для этого нужно добавить в `SpitterController` еще один метод-обработчик:

```
@RequestMapping(value="/{username}", method=RequestMethod.GET)
public String showSpitterProfile(@PathVariable String username,
        Model model) {
    model.addAttribute(spitterService.getSpitter(username));
    return "spitters/view";
}
```

Метод `showSpitterProfile()` не слишком отличается от других методов-обработчиков, созданных до сих пор. Он принимает строковый параметр с именем пользователя и с его помощью извлекает объект `Spitter` из базы данных. Затем он помещает найденный объект `Spitter` в модель и возвращает логическое имя представления, которое отобразит страницу с настройками.

Однако метод `showSpitterProfile()` имеет ряд отличительных особенностей. Во-первых, атрибут `value` в аннотации `@RequestMapping` содержит странные фигурные скобки. А во-вторых, параметр `username` отмечен аннотацией `@PathVariable`.

Вместе эти две особенности позволяют методу `showSpitterProfile()` обрабатывать запросы к адресам URL с параметрами в их путях. Фрагмент пути `{username}` фактически является меткой-заполнителем, соответствующей параметру `username` метода, отмеченному аннотацией `@PathVariable`. Независимо от того, что находится в соответствующем фрагменте пути, эта часть строки будет передана методу в виде параметра `username`.

Например, если запрос выполнен по URL с путем `/username/habuma`, тогда в параметре `username` методу `showSpitterProfile()` будет передано имя пользователя «`habuma`».

Подробнее об аннотации `@PathVariable` и о том, как она помогает создавать методы-обработчики для URL в стиле RESTful, рассказывается в главе 11.

Но реализация обработки формы в методе `addSpitterFromForm()` еще не закончена. Вероятно, вы помните, что параметр `Spitter` метода `addSpitterFromForm()` отмечен аннотацией `@Valid`. Поэтому посмотрим, как используется эта аннотация для сохранения данных, полученных в составе формы, отправленной пользователем.

8.4.3. Проверка входных данных

К процедуре регистрации нового пользователя предъявляются определенные требования. В частности, новый пользователь должен указать свое полное имя, адрес электронной почты, имя учетной

записи и пароль. Но и это еще не все, адрес электронной почты не может иметь произвольный формат – он должен быть похожим на адрес электронной почты. Кроме того, пароль должен содержать как минимум шесть символов.

Аннотация `@Valid` – первая линия обороны от недопустимых входных данных. В действительности аннотация `@Valid` является частью спецификации «JavaBean Validation API»¹. Версия Spring 3 включает поддержку JSR-303, и мы воспользовались аннотацией `@Valid`, чтобы сообщить фреймворку Spring, что объект `Spitter` должен быть проверен, так как значения своих свойств он получает из данных, введенных пользователем.

Если в процессе проверки объекта `Spitter` будут обнаружены недопустимые данные, информация об ошибках будет передана методу `addSpitterFromForm()` в виде объекта `BindingResult` во втором параметре. Если метод `BindingResult.hasErrors()` вернет `true`, это означает, что проверка потерпела неудачу. В этом случае метод вернет логическое имя представления `spitters/edit`, чтобы обеспечить повторное отображение формы и дать пользователю возможность исправить ошибки.

Но как Spring отличит допустимый объект `Spitter` от недопустимого?

Объявление правил проверки

Кроме всего прочего, спецификация JSR-303 определяет несколько аннотаций, определяющих правила проверки допустимости значений в свойствах. Чтобы определить «допустимость» каждого свойства объекта `Spitter`, можно использовать эти аннотации. В листинге 8.11 представлены свойства класса `Spitter`, отмеченные аннотациями с правилами проверки.

Листинг 8.11. Свойства класса Spitter, отмеченные аннотациями с правилами проверки

```
@Size(min=3, max=20,
       message="Username must be between 3 and 20 characters long.")

@Pattern(regexp="^*[a-zA-Z0-9]+*$",
          message="Username must be alphanumeric with no spaces")
private String username;
```

¹ Так же известной как JSR-303 (<http://jcp.org/en/jsr/summary?id=303>).

```

@Size(min=6, max=20,
      message="The password must be at least 6 characters long.")
private String password;

@Size(min=3, max=50,
      message="Your full name must be between 3 and 50 characters long.")
private String fullName;

@Pattern(regexp="[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,4}",
      message="Invalid email address.")
private String email;

```

Первые три свойства в листинге 8.11 отмечены аннотацией `@Size`, определяемой спецификацией JSR-303, реализующей проверку длины значения аннотированного свойства. Свойство `username` должно содержать не менее 3 и не более 20 символов, тогда как свойство `fullName` должно иметь длину от 3 до 50 символов. Свойство `password` должно содержать не менее 6 символов и не более 20.

Чтобы убедиться, что значение свойства `email` формально соответствует формату адреса электронной почты, оно было отмечено аннотацией `@Pattern` с регулярным выражением в атрибуте `regexp`, на соответствие которому должно проверяться значение свойства¹. Аналогично с помощью аннотации `@Pattern` проверяется, что свойство `username` содержит только алфавитные символы без пробелов.

Во всех аннотациях выше, реализующих проверку, определяется атрибут `message` с текстом сообщения, отображаемым в форме, в случае, когда проверка терпит неудачу, чтобы пользователь знал, какое значение следует исправить.

Теперь, когда пользователь отправит форму регистрации методу `addSpitterFromForm()` контроллера `SpitterController`, значения полей объекта `Spitter` будут проверены в соответствии с правилами в аннотациях. Если какое-либо из правил будет нарушено, метод-обработчик вновь отправит пользователя к форме ввода, чтобы он мог исправить ошибки.

При повторной отправке формы пользователю необходимо иметь некоторый механизм, который позволил бы описать проблему. Поэтому вернемся к JSP-файлу с определением формы и добавим код, отображающий сообщения механизма проверки.

¹ Имейте в виду, что здесь проверяется не сам адрес электронной почты, а лишь его соответствие определенному формату.

Отображение сообщений механизма проверки

Напомню, что объект `BindingResult`, переданный методу `addSpitterFromForm()`, позволяет определить, были ли выявлены ошибки в процессе проверки формы. Для этого достаточно просто вызвать его метод `hasErrors()`. Но в этом объекте также содержатся сообщения об ошибках для полей, не прошедших проверку.

Один из способов отобразить текст сообщений об ошибках состоит в том, чтобы извлекать сообщения об ошибках для отдельных полей вызовом метода `getFieldError()` объекта `BindingResult`. Но в Spring имеется более удобный способ – с использованием библиотеки тегов JSP. В частности, ошибки, выявленные при проверке, позволяет отобразить тег `<sf:errors>`. Для этого достаточно добавить несколько тегов `<sf:errors>` в определение формы, как показано в листинге 8.12.

Листинг 8.12. Тег `<sf:errors>` можно использовать для отображения сообщений об ошибках

```
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form"%>

<div>
<h2>Create a free Spitter account</h2>

<sf:form method="POST" modelAttribute="spitter"
          enctype="multipart/form-data">
<fieldset>
<table cellspacing="0">
<tr>
    <th><sf:label path="fullName">Full name:</sf:label></th>
    <td><sf:input path="fullName" size="15" /><br/>
        <sf:errors path="fullName" cssClass="error" /> <!-- Сообщить об -->
        <!-- ошибке в поле fullName -->
    </td>
</tr>
<tr>
    <th><sf:label path="username">Username:</sf:label></th>
    <td><sf:input path="username" size="15" maxlength="15" />
        <small id="username_msg">No spaces, please.</small><br/>
        <sf:errors path="username" cssClass="error" /> <!-- Сообщить об -->
        <!-- ошибке в поле username -->
    </td>
</tr>
<tr>
    <th><sf:label path="password">Password:</sf:label></th>
    <td><sf:password path="password" size="30" />
        <sf:errors path="password" cssClass="error" /> <!-- Сообщить об -->
        <!-- ошибке в поле password -->
    </td>
</tr>
```

```
        showPassword="true"/>
    <small>6 characters or more (be tricky!)</small><br/>
    <sf:errors path="password" cssClass="error" />      <!-- Сообщить об --&gt;
    &lt;/td&gt;                                              &lt!-- ошибке в поле password --&gt;
&lt;/tr&gt;
&lt;tr&gt;
    &lt;th&gt;&lt;sf:label path="email"&gt;Email Address:&lt;/sf:label&gt;&lt;/th&gt;
    &lt;td&gt;&lt;sf:input path="email" size="30"/&gt;
        &lt;small&gt;In case you forget something&lt;/small&gt;&lt;br/&gt;
        &lt;sf:errors path="email" cssClass="error" /&gt;      <!-- Сообщить об --&gt;
    &lt;/td&gt;                                              &lt!-- ошибке в поле email --&gt;
&lt;/tr&gt;
&lt;tr&gt;
    &lt;th&gt;&lt;/th&gt;
    &lt;td&gt;
        &lt;sf:checkbox path="updateByEmail"/&gt;
        &lt;sf:label path="updateByEmail"&gt;
            Send me email updates!&lt;/sf:label&gt;
        &lt;/td&gt;
&lt;/tr&gt;
&lt;tr&gt;
    &lt;th&gt;&lt;label for="image"&gt;Profile image:&lt;/label&gt;&lt;/th&gt;
    &lt;td&gt;&lt;input name="image" type="file"/&gt;
&lt;/tr&gt;
&lt;tr&gt;
    &lt;th&gt;&lt;/th&gt;
    &lt;td&gt;&lt;input name="commit" type="submit"
        value="I accept. Create my account." /&gt;&lt;/td&gt;
    &lt;/tr&gt;
&lt;/table&gt;
&lt;/fieldset&gt;
&lt;/sf:form&gt;
&lt;/div&gt;</pre>
```

Атрибут `path` элемента `<sf:errors>` определяет поле формы, рядом с которым следует вывести сообщение об ошибке. Например, следующий элемент `<sf:errors>` обеспечивает вывод сообщения об ошибке (если таковая имеется) для поля с именем `fullName`:

```
<sf:errors path="fullName" cssClass="error" />
```

Если в значении поля будет обнаружено несколько ошибок, будут выведены все соответствующие сообщения, разделенные HTML-тегом `
`. Если потребуется отделять сообщения друг от друга

каким-то иным способом, для этого можно использовать атрибут delimiter. Следующий элемент <sf:errors> выводит сообщения об ошибках через запятую и пробел:

```
<sf:errors path="fullName" delimiter=", " cssClass="error" />
```

Обратите внимание, что в этой JSP-странице присутствуют четыре тега <sf:errors> – по одному для каждого поля, где объявлены правила проверки. Атрибут cssClass ссылается на класс CSS, чтобы обеспечить вывод сообщений об ошибках красным цветом для привлечения внимания пользователя.

Теперь если при проверке будут обнаружены какие-либо ошибки, они появятся на странице с формой. Например, на рис. 8.7 показано, как будет выглядеть форма, если пользователь оставит все поля незаполненными.

Как показано на рис. 8.7, сообщения об ошибках отображаются отдельно для каждого поля формы. Но если потребуется отобразить все сообщения об ошибках в одном месте (например, вверху

The screenshot shows a registration form titled "Create a free Spitter account". It includes fields for Full name, Username, Password, Email Address, and a checkbox for email updates. Below each field is a red error message indicating a validation error: "Your full name must be between 3 and 50 characters long.", "Username must be between 3 and 20 characters long.", "The password must be at least 6 characters long.", and "Invalid email address.". A note "In case you forget something" is present next to the email field. A "Send me email updates!" checkbox is also shown. At the bottom is a button labeled "I accept. Create my account."

Create a free Spitter account

Full name:

Your full name must be between 3 and 50 characters long.

Username: No spaces, please.

Username must be between 3 and 20 characters long.

Password: 6 characters or more (be tricky!)

The password must be at least 6 characters long.

Email Address: In case you forget something

Invalid email address.

Send me email updates!

I accept. Create my account.

Рис. 8.7. JSP-теги <sf:errors> в форме регистрации обеспечивают отображение сообщений об ошибках, чтобы пользователь мог исправить их и повторить попытку

формы), достаточно будет добавить единственный тег `<sf:errors>` с атрибутом `path`, имеющим значение `*`:

```
<sf:errors path="*" cssClass="error" />
```

Теперь вы знаете, как писать методы-обработчики контроллеров, обрабатывающие данные формы. Но все эти поля форм, обсуждавшиеся выше, объединяет одно обстоятельство – они являются текстовыми полями ввода и предназначены для ввода данных с клавиатуры. А как быть, если пользователю потребуется представить в форме некоторые данные, которые невозможно ввести с клавиатуры? Например, отправить изображение или файл какого-то другого типа?

8.5. Выгрузка файлов

В разделе 8.2.4, выше, отмечалось, что в качестве аватара любого пользователя отображается изображение по умолчанию `spitter-avatar.png`. Но настоящие пользователи приложения Spitter наверняка предпочли бы иметь изображения, подчеркивающие индивидуальность. Чтобы дать им такую возможность, реализуем выгрузку аватара пользователя как часть процедуры регистрации.

Чтобы обеспечить выгрузку файлов, в приложении Spitter необходимо соблюсти три условия:

- добавить в форму поле для выгрузки файла;
- добавить в метод `addSpitterFromForm()` контроллера `SpitterController` поддержку приема выгруженного файла;
- настроить в Spring механизм анализа форм, состоящих из нескольких частей.

Начнем с первого пункта списка и подготовим форму к приему выгружаемого файла.

8.5.1. Добавление в форму поля выгрузки файла

Большинство полей ввода в форме являются текстовыми и легко могут быть отправлены на сервер в виде множества пар имя/значение. В действительности *тип содержимого* типичной формы определяется как `application/x-wwwform-urlencoded` и имеет форму строки с парами имя/значение, разделенными символами амперсанда.

Но, я думаю, вы согласитесь, что файлы существенно отличаются от значений других полей формы. Обычно на сервер выгружаются двоичные файлы, что не укладывается в парадигму пар имя/значение. Таким образом, чтобы дать пользователям возможность выгружать изображения, играющие роль аватара, необходимо представить данные формы в каком-то другом виде.

Что касается отправки форм с файлами, их тип определяется как `multipart/form-data`. То есть нам необходимо настроить тип содержимого формы `multipart/form-data`, определив атрибут `enctype` в элементе `<sf:form>`, как показано ниже:

```
<sf:form method="POST"
    modelAttribute="spitter"
    enctype="multipart/form-data">
```

После установки атрибута `enctype` в значение `multipart/form-data` каждое поле будет отправляться как отдельная часть запроса POST, а не как пара имя/значение. Это делает возможным в одной из частей отправить содержимое файла изображения.

Теперь можно добавить в форму новое поле – стандартный HTML-тег `<input>` с атрибутом `type`, имеющим значение `file`:

```
<tr>
    <th><label for="image">Profile image:</label></th>
    <td><input name="image" type="file"/>
</tr>
```

Этот фрагмент HTML-разметки отобразит поле выбора файла в форме. В большинстве браузеров это поле отображается как текстовое поле с кнопкой сбоку. На рис. 8.8 показано, как выглядит это поле при отображении формы в браузере Safari в Mac OS X.

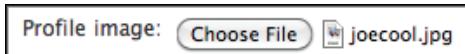


Рис. 8.8. Поле выбора файла в форме регистрации приложения Spitter, позволяющее пользователю выбрать индивидуальный аватар

Теперь пользователи смогут выгружать файлы с изображениями своих аватаров. Такая форма будет отправляться на сервер как состоящая из нескольких частей, где одна из них будет содержать

двоичные данные из файла изображения. Следующий наш шаг – подготовить серверную часть приложения к приему этих данных.

8.5.2. Прием выгруженных файлов

Как и прежде, данные формы будет обрабатывать метод `addSpitterFromForm()`. Но нам нужно добавить в него обработку выгруженного изображения. В листинге 8.13 представлена новая версия метода `addSpitterFromForm()`, принимающая выгруженное изображение.

Листинг 8.13. Метод `addSpitterFromForm()`, принимающий параметр типа `MultipartFile`

```
@RequestMapping(method=RequestMethod.POST)
public String addSpitterFromForm(@Valid Spitter spitter,
        BindingResult bindingResult,
        @RequestParam(value="image", required=false)           // Прием файла
        MultipartFile image) {

    if(bindingResult.hasErrors()) {
        return "spitters/edit";
    }

    spitterService.saveSpitter(spitter);

    try {
        if(!image.isEmpty()) {
            validateImage(image);                         // Проверить изображение

            saveImage(spitter.getId() + ".jpg", image); // Сохранить файл
        }
    } catch (ImageUploadException e) {
        bindingResult.reject(e.getMessage());
        return "spitters/edit";
    }

    return "redirect:/spitters/" + spitter.getUsername();
}
```

Первое изменение в методе `addSpitterFromForm()` касается добавления нового параметра. Параметр `image` имеет тип `MultipartFile` и отмечен аннотацией `@RequestParam`, указывающей, что этот параметр не является обязательным (то есть пользователь может не отправлять изображение аватара при регистрации).

Чуть ниже в этом методе проверяется наличие изображения и, если оно имеется, – передается сначала методу validateImage(), а затем методу saveImage(). Метод validateImage(), представленный ниже, проверяет соответствие файла изображения предъявляемым требованиям:

```
private void validateImage(MultipartFile image) {  
    if(!image.getContentType().equals("image/jpeg")) {  
        throw new ImageUploadException("Only JPG images accepted");  
    }  
}
```

Мы не должны позволять пользователям выгружать файлы .zip или .exe. Поэтому метод validateImage() проверяет, является ли выгруженный файл изображением в формате JPEG. Если файл не соответствует требованиям, возбуждается исключение ImageUploadException (расширяющее исключение RuntimeException).

Убедившись, что выгруженный файл является изображением, его можно сохранить, вызвав метод saveImage(). Фактически метод saveImage() мог бы сохранить изображение в любом месте, доступном для браузера. Чтобы не усложнять, для начала напишем реализацию метода saveImage(), сохраняющую изображение в локальной файловой системе.

Сохранение файлов в файловой системе

Даже при том, что доступ к нашему приложению будет осуществляться через сеть, его ресурсы в конечном итоге будут храниться в *файловой системе* сервера. Поэтому вполне естественным сохранять пользовательские файлы с изображениями в локальной файловой системе, там, где они будут доступны веб-серверу. Именно это и делает следующая реализация метода saveImage:

```
private void saveImage(String filename, MultipartFile image)  
    throws ImageUploadException {  
    try {  
        File file = new File(webRootPath + "/resources/" + filename);  
        FileUtils.writeByteArrayToFile(file, image.getBytes());  
    } catch (IOException e) {  
        throw new ImageUploadException("Unable to save image", e);  
    }  
}
```

Сначала метод `saveImage()` создает объект `java.io.File`, определяя путь к файлу, исходя из значения `webRootPath`. Здесь преднамеренно не раскрывается значение этой переменной, так как оно зависит от сервера, где выполняется приложение. Достаточно будет сказать, что получение этого значения можно организовать через внедрение либо с помощью метода `setWebRootPath()`, либо за счет чтения значения из конфигурационного файла с помощью выражения на языке SpEL и аннотации `@Value`.

После подготовки объекта `File` изображение записывается в файл с помощью `FileUtils` из Apache Commons IO¹. Если что-то пойдет не так, будет возбуждено исключение `ImageUploadException`.

Хранение файлов изображений в файловой системе прекрасно подходит для целей приложения, но влечет за собой необходимость следить за файловой системой. Вам придется гарантировать наличие достаточного свободного пространства, создавать резервные копии на случай отказа аппаратных средств, а также обеспечить синхронизацию файлов изображений между несколькими серверами в классике.

Другой вариант заключается в том, чтобы позволить кому-то другому избавить вас от всех этих неприятностей. Добавив немногого программного кода, можно организовать хранение изображений в облаке. Давайте освободим себя от необходимости управлять файлами изображений, переписав метод `saveFile()` так, чтобы он сохранял их в хранилище Amazon S3.

Сохранение файлов в хранилище Amazon S3

Служба *Simple Storage Service* компании Amazon, или S3, как ее часто называют, представляет собой недорогой способ хранения файлов в инфраструктуре серверов компании Amazon. Используя службу S3, можно просто передавать ей файлы и переложить всю рутинную работу на системных администраторов Amazon.

Самый простой способ взаимодействий со службой S3 в программах на языке Java обеспечивает библиотека JetS3t². JetS3t – это открытая библиотека функций, реализующих запись и чтение файлов в облаке S3. Мы можем задействовать библиотеку JetS3t для сохранения аватаров пользователей. В листинге 8.14 представлена новая версия метода `saveImage()`.

¹ <http://commons.apache.org/io/>.

² <http://bitbucket.org/jmurty/jets3t/wiki/Home>.

Листинг 8.14. Версия метода saveImage(), сохраняющая файлы изображений в облаке Amazon S3

```
private void saveImage(String filename, MultipartFile image)
    throws ImageUploadException {
    try {
        AWSredentials awsCredentials =
            new AWSredentials(s3AccessKey, s3SecretKey);
        S3Service s3 = new RestS3Service(awsCredentials); // Настройка S3

        // Создать объекты, представляющие хранилище и изображение
        S3Bucket imageBucket = s3.getBucket("spitterImages");
        S3Object imageObject = new S3Object(filename);

        // Скопировать данные изображения в объект
        imageObject.setInputStream(
            new ByteArrayInputStream(image.getBytes()));
        imageObject.setContentLength(image.getBytes().length);
        imageObject.setContentType("image/jpeg");

        // Определить разрешения
        AccessControlList acl = new AccessControlList();
        acl.setOwner(imageBucket.getOwner());
        acl.grantPermission(GroupGrantee.ALL_USERS,
            Permission.PERMISSION_READ);
        imageObject.setAcl(acl);

        // Сохранить изображение
        s3.putObject(imageBucket, imageObject);
    } catch (Exception e) {
        throw new ImageUploadException("Unable to save image", e);
    }
}
```

В первую очередь метод saveImage() настраивает параметры аутентификации в службе Amazon Web Service. Для этого необходимо иметь ключ доступа к службе S3 и секретный ключ S3. Их выдает компания Amazon после подписки на пользование услугами службы S3. Они передаются контроллеру SpitterController посредством механизма внедрения.

После настройки параметров аутентификации метод saveImage() создает экземпляр объекта RestS3Service, посредством которого бу-

дут осуществляться взаимодействия с файловой системой S3. Затем он получает ссылку на хранилище `spitterImages`, создает объект `S3Object` для изображения и затем копирует в этот объект данные изображения.

Непосредственно перед вызовом метода `putObject()`, выполняющее запись изображения в S3, метод `saveImage()` устанавливает в объекте `S3Object` разрешения, позволяющие пользователям просматривать его. Это очень важно – в противном случае изображения будут недоступны пользователям нашего приложения.

Как и в предыдущей версии метода `saveImage()`, в случае каких-либо проблем возбуждается исключение `ImageUploadException`.

Мы практически закончили реализацию выгрузки файлов изображений. Осталось лишь связать все вместе.

8.5.3. Настройка Spring для выгрузки файлов

Сам по себе контроллер `DispatcherServlet` понятия не имеет, что делать с формами, состоящими из нескольких частей. Поэтому мы должны реализовать механизм извлечения данных из запросов POST, чтобы контроллер `DispatcherServlet` смог передать их нашему контроллеру.

Чтобы зарегистрировать этот механизм в Spring, необходимо объявить компонент, реализующий интерфейс `MultipartResolver`. Выбор реализации не составляет затруднений, так как в состав фреймворка Spring входит только одна: `CommonsMultipartResolver`. Ниже показано, как настроить этот механизм в Spring:

```
<bean id="multipartResolver" class=
    "org.springframework.web.multipart.commons.CommonsMultipartResolver"
    p:maxUploadSize="500000" />
```

Имейте в виду, что идентификатор этого компонента играет очень важную роль. Когда контроллеру `DispatcherServlet` потребуется задействовать механизм извлечения частей формы, он будет искать компонент с идентификатором `multipartResolver`. Если искомый компонент будет иметь другой идентификатор, `DispatcherServlet` просто не найдет его.

8.6. Альтернативы JSP¹

В октябре 1908 года Генри Форд (Henry Ford) выпустил «автомобиль для широких масс»: Форд Модель-Т (Model-T Ford). Прейскурантная цена составила \$950. Чтобы ускорить сборку, все автомобили серии «Модель-Т» красились в черный цвет, потому что черная краска сохла быстрее остальных. Легенда приписывает Генри Форду слова: «Вы можете купить у меня автомобиль любого цвета, при условии что этот цвет будет черным».

С тех пор автомобили проделали длинный путь в своем развитии. В дополнение к головокружительному разнообразию форм кузова у нас появилась также возможность выбирать модель радиоприемника/CD-проигрывателя, заказывать установку электрических стеклоподъемников и замков на двери, а также матерчатую или кожаную обивку салона. И в наше время любой может купить автомобиль любого цвета, включая черный.

До настоящего момента мы использовали технологию JSP при создании представлений для приложения. Но, в отличие от черной краски Генри Форда, JSP – не единственная технология создания представлений, доступная для выбора. Двумя другими популярными механизмами шаблонов являются Velocity и FreeMarker. Посмотрим, как использовать эти механизмы вместе с фреймворком Spring MVC.

8.6.1. Использование шаблонов Velocity

Velocity – простой в использовании язык шаблонов для приложений на Java. Шаблоны Velocity не содержат программного кода на Java, что облегчает их понимание не только разработчиками, но и неразработчиками. Цитата из руководства пользователя Velocity: «Velocity отделяет код Java от веб-страниц, делая веб-сайт более простым для длительного сопровождения и обеспечивая жизнеспособную альтернативу технологии JavaServer Pages».

¹ Этот и следующий разделы взяты из предыдущей редакции книги, охватывающей Spring 2. В связи с этим хотелось бы отметить следующее: несмотря на то что в SpringSource стараются поддерживать обратную совместимость между версиями продукта, есть небольшой риск, что конкретно этот функционал может работать несколько иначе в версии Spring 3. Поэтому информация в этих разделах актуальна для версии Spring 2. – *Прим. ред.*

После JSP Velocity – вероятно, самый популярный язык шаблонов для веб-приложений на основе Java. Поэтому вполне возможно, что у вас появится желание разработать Spring-приложение с использованием Velocity в качестве технологии реализации уровня представлений. К счастью, Spring поддерживает Velocity как язык шаблонов уровня представлений для Spring MVC.

Посмотрим, как использовать Velocity со Spring MVC, реализовав уровень представлений приложения RoadRantz на основе Velocity.

Определение представления средствами Velocity

Предположим, что в качестве технологии реализации уровня представлений вы решили использовать Velocity вместо JSP. Выше уже был представлен пример реализации домашней JSP-страницы. А теперь посмотрим на шаблон Velocity – `home.vm` (листинг 8.15), используемый для отображения домашней страницы.

Листинг 8.15. Реализация домашней страницы на основе Velocity

```
<html>
    <head><title>Rantz</title></head>

    <body>
        <h2>Rantz:</h2>

        <a href="addRant.htm">Add rant</a><br/>
        <a href="register.htm">Register new motorist</a><br/>
        <ul>
            #foreach($rant in $rants)           <!-- Итерации по элементам списка --&gt;
            &lt;li&gt; ${rant.vehicle.state} /
                ${rant.vehicle.plateNumber} --
                ${rant.rantText}&lt;/li&gt;
            #end
        &lt;/ul&gt;
    &lt;/body&gt;
&lt;/html&gt;</pre>

---


```

Шаблоны Velocity и JSP имеют не так много отличий. Но есть одно, сразу бросающееся в глаза, – отсутствие тегов шаблона. Это обусловлено тем, что механизм Velocity не опирается на использование тегов, как JSP. Вместо этого используется собственный язык, известный как Velocity Template Language (VTL), содержащий инструкции управления потоком выполнения и другие директивы.

Директива `#foreach` в файле `home.vm` используется для организации цикла по списку элементов, отображая в каждой итерации свойства очередного элемента.

Несмотря на это отличие, выражения языка Velocity во многом напоминают язык JSP. Фактически JSP просто следовал по пятам за Velocity, используя нотацию `${}` в выражениях своего собственного языка.

Приведенный пример шаблона демонстрирует лишь малую толику возможностей Velocity. Чтобы узнать больше, посетите домашнюю страницу проекта Velocity по адресу: <http://jakarta.apache.org/velocity>.

Теперь, после создания шаблона, необходимо настроить Spring на использование шаблонов Velocity в качестве представления в приложении на основе Spring MVC.

Настройка механизма Velocity

В первую очередь следует настроить сам механизм Velocity. Для этого нужно объявить компонент `VelocityConfigurer` в файле конфигурации Spring, как показано ниже:

```
<bean id="velocityConfigurer"
      class="org.springframework.web.servlet.view.velocity.
      ↳ VelocityConfigurer">
    <property name="resourceLoaderPath" value="WEB-INF/velocity/" />
</bean>
```

Компонент `VelocityConfigurer` отвечает за настройку механизма Velocity в Spring. Свойство `resourceLoaderPath` определяет место поиска шаблонов. Я рекомендую помещать шаблоны в каталог, находящийся внутри каталога WEB-INF, чтобы к шаблонам нельзя было обратиться непосредственно.

Знакомые с Velocity уже знают, что настройки поведения Velocity можно определять в файле `velocity.properties`. Однако те же самые настройки можно определить с помощью `VelocityConfigurer`, в свойстве `velocityProperties`. Например, взгляните на следующее объявление компонента `VelocityConfigurer`:

```
<bean id="velocityConfigurer"
      class="org.springframework.web.servlet.view.velocity.
      ↳ VelocityConfigurer">
```

```
<property name="resourceLoaderPath" value="WEB-INF/velocity/" />
<property name="velocityProperties">
    <props>
        <prop key="directive.foreach.counter.name">loopCounter</prop>
        <prop key="directive.foreach.counter.initial.value">0</prop>
    </props>
</property>
</bean>
```

Здесь мы настроили механизм Velocity и изменили поведение цикла `#foreach`. По умолчанию цикл `#foreach` поддерживает переменную-счетчик с именем `$velocityCount`, которая получает значение 1 в первой итерации цикла. Здесь мы присвоили свойству `directive.foreach.counter.name` значение `loopCounter`, чтобы к счетчику цикла можно было обращаться по имени `$loopCounter`. Кроме того, мы определили начальное значение счетчика цикла, присвоив значение 0 свойству `directive.foreach.counter.initial.value`. (Более подробную информацию о конфигурационных свойствах Velocity можно получить в руководстве разработчика по адресу: <http://jakarta.apache.org/velocity/developer-guide.html>).

Разрешение представлений Velocity

Последнее, что необходимо сделать перед использованием шаблонов представлений Velocity, – настроить арбитр представлений. В частности, объявитите компонент `VelocityViewResolver` в файле конфигурации контекста, как показано ниже:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
      ↗ velocity.VelocityViewResolver">
    <property name="suffix" value=".vm" />
</bean>
```

Компонент `VelocityViewResolver` для Velocity – почти то же самое, что `InternalResourceViewResolver` для JSP. Как и `InternalResourceViewResolver`, он имеет свойства `prefix` и `suffix`, которые вместе с логическим именем представления используются для формирования пути к шаблону. В данном случае достаточно установить только свойство `suffix`, указав в нем расширение `.vm`. Префикс не требуется, потому что путь к каталогу шаблонов уже был указан в свойстве `resourceLoaderPath` компонента `VelocityConfigurer`.

Примечание. Здесь атрибуту `id` компонента присвоено значение `viewResolver`. Это обстоятельство играет важную роль, когда сервлет `DispatcherServlet` не настроен на поиск всех арбитров представлений. Если в приложении придется использовать несколько арбитров представлений, то, вероятно, придется изменить значение атрибута `id` на что-то более подходящее (и уникальное), такое как `velocityViewResolver`.

Теперь приложение готово отображать представления, основанные на шаблонах Velocity. Для этого достаточно просто вернуть объект `ModelAndView`, ссылающийся на представление по его логическому имени. В случае с `HomeController` делать вообще ничего не потребуется, потому что он уже возвращает объект `ModelAndView`:

```
return new ModelAndView("home", "rants", recentRants);
```

Представление имеет логическое имя `home`. Когда имя представления будет определено, к `home` будет прибавлено расширение `.vm`, чтобы сконструировать имя файла шаблона `home.vm`. Компонент `VelocityViewResolver` найдет этот шаблон в пути `WEB-INF/velocity/`.

Что касается объекта модели `rants`, то он будет доступен в шаблоне в виде свойства Velocity. В листинге 8.15 этот объект представляет коллекцию, по которой директива `#foreach` выполняет итерации.

Форматирование дат и чисел

Несмотря на то что приложение уже готово отображать представления Velocity, осталось еще несколько нерешенных вопросов. Если внимательно изучить шаблон `home.vm` (листинг 8.15), можно заметить, что в `home.vm` отсутствует форматирование дат, поэтому было бы желательно настроить некоторые параметры и тем самым обеспечить надлежащее форматирование даты.

Язык VTL не поддерживает форматирование дат непосредственно. Однако в Velocity имеются инструменты для форматирования дат и чисел. Чтобы сделать их доступными, необходимо сообщить арбитру `VelocityViewResolver` названия атрибутов, посредством которых эти инструменты будут экспортироваться. Имена атрибутов определяются через свойства `dateToolAttribute` и `numberToolAttribute` компонента `VelocityViewResolver`:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
      ↳ velocity.VelocityViewResolver">
```

```
<property name="suffix" value=".vm" />
<property name="dateToolAttribute">
    <value>dateTool</value>
</property>
<property name="numberToolAttribute">
    <value>numberTool</value>
</property>
</bean>
```

Здесь указывается, что инструмент форматирования дат будет доступен в шаблоне под именем \$dateTool. То есть, чтобы отформатировать дату, достаточно передать ее функции format инструмента. Например:

```
$dateTool.format("FULL", rant.postedDate)
```

Первый параметр – строка шаблона. Здесь используется тот же синтаксис, что и в java.text.SimpleDateFormat. Кроме того, можно указать один из стандартных шаблонов java.text.DateFormat, передав в виде строки шаблона одно из значений: FULL, LONG, MEDIUM, SHORT или DEFAULT. Здесь использовано значение FULL, соответствующее формату представления полной даты.

Как уже упоминалось, атрибут \$numberTool представляет в шаблоне Velocity инструмент для форматирования чисел. За более подробной информацией об этом инструменте и об инструменте форматирования дат обращайтесь к документации Velocity.

Экспортирование атрибутов запроса и сеанса

Большинство данных, которые должны быть отображены в шаблоне Velocity, можно передать представлению через объект Map, который передается объекту ModelAndView, однако иногда может потребоваться вывести значения атрибутов таких объектов сервлета, как запрос или сеанс. Например, если пользователь зарегистрировался в приложении, в объекте сеанса сервлета может содержаться информация о пользователе.

Было бы очень неудобно, если бы мы были вынуждены копировать атрибуты из запроса или сеанса в данные модели в каждом контроллере. К счастью, VelocityViewResolver может копировать атрибуты в модель автоматически. Свойства exposeRequestAttributes и exposeSessionAttributes сообщают компоненту VelocityViewResolver о необходимости копирования атрибутов запроса и сеанса в модель данных. Например:

```
<bean id="viewResolver"
      class="org.springframework.
        ↗ web.servlet.view.velocity.VelocityViewResolver">

    ...
    <property name="exposeRequestAttributes">
        <value>true</value>
    </property>
    <property name="exposeSessionAttributes">
        <value>true</value>
    </property>
</bean>
```

По умолчанию оба эти свойства имеют значение `false`. Но здесь мы установили их в значение `true`, так как нам необходимо, чтобы атрибуты запроса и сеанса были скопированы в модель данных и были доступны в шаблоне Velocity.

Связывание полей форм в шаблонах Velocity

Ранее в этой главе было показано, как с помощью библиотеки тегов JSP в Spring связывать поля форм со свойствами объектов и отображать сообщения об ошибках. Хотя в Velocity нет тегов, тем не менее Spring обеспечивает ряд макроопределений Velocity, представляющих возможности, эквивалентные библиотекам тегов JSP. В табл. 8.2 перечислены макроопределения Velocity, поставляемые в составе Spring.

Большая часть макроопределений в табл. 8.2 предназначена для связывания полей форм. То есть они генерируют элементы HTML-форм, значения которых связаны со свойствами объектов. Конкретное свойство объекта для связывания определяется в первом параметре (с именем `path` в таблице). Большинство макроопределений имеют параметр, позволяющий определять дополнительные атрибуты для генерируемых HTML-элементов (например, `length="20"`).

Таблица 8.2. Spring MVC предоставляет коллекцию макроопределений Velocity для связывания полей форм с объектом

Макроопределение	Назначение
<code>#springFormCheckboxes(path options separator attributes)</code>	Отображает набор флажков. Устанавливает состояние флажков согласно значениям свойств объекта
<code>#springFormHiddenInput(path attributes)</code>	Генерирует скрытое поле, связанное со свойством объекта

Таблица 8.2 (окончание)

Макроопределение	Назначение
#springFormInput(path attributes)	Отображает текстовое поле, связанное со свойством объекта
#springFormMultiSelect(path options attributes)	Отображает список выбора, допускающий возможность выбора нескольких пунктов одновременно, связанный со свойством объекта
#springFormPasswordInput(path attributes)	Отображает поле ввода пароля, связанное со свойством объекта
#springFormRadioButtons(path options separator attributes)	Отображает набор радиокнопок, связанный со свойством объекта
#springFormSingleSelect(path options attributes)	Отображает список выбора, допускающий возможность выбора единственного пункта, связанный со свойством объекта. Выбранное значение связано со свойством объекта
#springFormTextarea(path attributes)	Отображает текстовую область ввода, связанную со свойством объекта
#springMessage(messageCode)	Отображает «внешнее сообщение» из пакета ресурсов
#springMessageText(messageCode text)	Отображает «внешнее сообщение» из пакета ресурсов со значением по умолчанию, если сообщение не найдено в пакете ресурсов
#springShowErrors(separator class/style)	Отображает ошибки, выявленные при проверке
#springUrl(relativeUrl)	Отображает абсолютный URL по заданному относительному URL

Для демонстрации использования макроопределений Velocity в Spring рассмотрим еще раз представление addRant. В листинге 8.16 приводится содержимое файла addRant.vm – шаблона Velocity этого представления.

Листинг 8.16. Шаблон Velocity представления для добавления сообщений

```

<html>
  <head>
    <title>#springMessage("title.addRant")</title>
  </head>

  <body>
    <h2>#springMessage("title.addRant")</h2>
    <form method="POST" action="addRant.htm">
      <b>#springMessage("field.state")</b>#springFormInput(

```

```
        "rant.vehicle.state" "")<br/>
<b>#springMessage("field.plateNumber") </b>
#springFormInput("rant.vehicle.plateNumber" "")<br/>
#springMessage("field.rantText")
#springFormTextarea("rant.rantText" "rows='5' cols='50'")
<input type="submit"/>
</form>
</body>
</html>
```

Для организации ввода названия страны и регистрационного номера используются макроопределения `#springFormInput`, связывающие поля ввода со свойствами `rant.vehicle.state` и `rant.vehicle.plateNumber` соответственно. Это означает, что при получении формы, значения будут свойства `state` и `plateNumber` свойства `vehicle` управляющего объекта (`rant`). В обоих случаях нет необходимости определять дополнительные атрибуты в разметке HTML, поэтому во втором параметре передается пустая строка. В результате получается следующая разметка HTML с полями:

```
<b>State: </b><input type="text" id="vehicle.state"
    name="vehicle.state" value="" /> <br/>
<b>Plate #: </b><input type="text" id="vehicle.plateNumber"
    name="vehicle.plateNumber" value="" /> <br/>
```

Далее следует область ввода `<textarea>`, куда пользователь вводит текст сообщения. Область ввода отображается макроопределением `#springFormTextarea`, которое связывает ее со свойством `rant.rantText`. То есть после получения формы текст будет сохранен в свойстве `rantText` управляющего объекта. В данном случае необходимо определить размеры текстовой области, поэтому во втором параметре мы передаем дополнительные атрибуты. Разметка HTML текстовой области выглядит следующим образом:

```
<textarea id="rantText" name="rantText"
    rows='5' cols='50'></textarea>
```

Чтобы иметь возможность использовать в своих шаблонах макроопределения из фреймворка Spring, необходимо включить их поддержку в свойстве `exposeSpringMacroHelpers` компонента `VelocityViewResolver`:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
      ↗ velocity.VelocityViewResolver">
    <property name="suffix" value=".vm" />
    <property name="exposeSpringMacroHelpers" value="true" />
</bean>
```

Установив свойство `exposeSpringMacroHelpers` в значение `true`, можно быть уверенными, что ваши шаблоны получат доступ к макроопределениям для Velocity, имеющимся в фреймворке Spring.

Механизм шаблонов Velocity широко используется как альтернатива JSP, но выбор альтернатив на этом не ограничивается. Другим широко известным механизмом шаблонов является FreeMarker, способный заменить JSP в приложениях на основе Spring MVC. Давайте посмотрим, как включить поддержку FreeMarker в приложение на основе Spring MVC.

8.6.2. Использование шаблонов **FreeMarker**

Механизм шаблонов FreeMarker является более сложным инструментом, чем Velocity, но эта сложность обусловлена более широкими возможностями. FreeMarker имеет встроенную поддержку некоторых задач, таких форматирование дат и чисел, и удаление пробельных символов. Эти возможности доступны в Velocity только через дополнительные инструменты.

Как вскоре будет показано, использование FreeMarker совместно с фреймворком Spring MVC не сильно отличается от использования Velocity. Но не будем забегать вперед и начнем с создания шаблона FreeMarker для использования в приложении.

Создание представления на основе FreeMarker

Допустим, что после дополнительных исследований был сделан вывод, что шаблоны FreeMarker лучше отвечают вашим потребностям, чем Velocity. Поэтому для реализации уровня представлений в приложении вместо Velocity было решено подключить FreeMarker к фреймворку Spring MVC. В листинге 8.17 представлен шаблон FreeMarker домашней страницы приложения `home.ftl`.

Листинг 8.17. Шаблон FreeMarker домашней страницы приложения

```
<html>
  <head><title>Rantz</title></head>
```

```
<body>
    <h2>Rantz:</h2>

    <a href="addRant.htm">Add rant</a><br/>
    <a href="register.htm">Register new motorist</a><br/>
    <ul>
        <#list rants as rant>
            <li>${rant.vehicle.state} /
                ${rant.vehicle.plateNumber} --
                ${rant.rantText}</li>
        </#list>
    </ul>
</body>
</html>
```

Как видите, версия домашней страницы на основе FreeMarker не сильно отличается от версии на основе Velocity, представленной в листинге 8.15. Как и в Velocity (и в JSP, если уж на то пошло), нотация \${} используется для отображения значений атрибутов.

Шаблон `home.ftl` демонстрирует лишь малую толику возможностей FreeMarker. Дополнительную информацию о механизме шаблонов FreeMarker можно найти на домашней странице проекта по адресу: <http://freemarker.sourceforge.net>.

Настройка механизма FreeMarker

Как и Velocity, механизм шаблонов FreeMarker требует настройки в контексте Spring для обеспечения возможности использования шаблонов FreeMarker в качестве представлений. Для начала необходимо объявить компонент `FreeMarkerConfigurer` в конфигурационном файле, как показано ниже:

```
<bean id="freemarkerConfig"
    class="org.springframework.web.servlet.view.
        ↗ freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="WEB-INF/freemarker/" />
</bean>
```

Компонент `FreeMarkerConfigurer` для FreeMarker суть то же самое, что компонент `VelocityConfigurer` для Velocity. Он используется для настройки механизма FreeMarker. Как минимум, нужно сообщить механизму FreeMarker, где находятся шаблоны. Для этого используется свойство `templateLoaderPath` (здесь указано, что шаблоны хранятся в каталоге `WEB-INF/freemarker/`).

Дополнительные параметры настройки FreeMarker определяются в виде свойств внутри свойства `freemarkerSettings`. Например, FreeMarker каждые пять секунд (по умолчанию) проверяет шаблоны на наличие изменений, чтобы при необходимости повторно загрузить их. Но проверка изменений в шаблонах может отнимать немалое время. Когда приложение находится в эксплуатации и не предполагается часто изменять шаблоны, интервал между проверками можно увеличить до часа или больше.

Для этого следует изменить параметр `template_update_delay` в свойстве `freemarkerSettings`. Например:

```
<bean id="freemarkerConfig"
      class="org.springframework.web.servlet.view.
      ↳ freemarker.FreeMarkerConfigurer">
    <property name="templateLoaderPath" value="WEB-INF/freemarker/" />
    <property name="freemarkerSettings">
      <props>
        <prop key="template_update_delay">3600</prop>
      </props>
    </property>
</bean>
```

Обратите внимание, что, подобно свойству `velocityProperties` компонента `VelocityConfigurer`, свойство `freemarkerSettings` содержит элемент `<props>`. В данном случае он содержит единственный элемент `<prop>` – параметр `template_update_delay` со значением 3600 (секунд), указывающий, что проверка на наличие обновлений будет выполняться один раз в час.

Разрешение представлений FreeMarker

Следующее, что необходимо сделать, – объявить арбитр представлений для FreeMarker:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
      ↳ freemarker.FreeMarkerViewResolver">
    <property name="suffix" value=".ftl" />
</bean>
```

Компонент `FreeMarkerViewResolver` действует подобно компонентам `VelocityViewResolver` и `InternalResourceViewResolver`. Разрешение имен

файлов шаблонов выполняется за счет добавления к логическому имени представления префикса, определяемого свойством `prefix`, и суффикса, определяемого свойством `suffix`. Как и при использовании `VelocityViewResolver`, здесь устанавливается только значение свойства `suffix`, потому что путь к шаблону уже определен в свойстве `templateLoaderPath` компонента `FreeMarkerConfigurer`.

Экспортирование атрибутов запроса и сеанса

В разделе 8.6.1 было показано, как настроить в компоненте `VelocityViewResolver` копирование атрибутов запроса и/или сеанса в модель, чтобы они были доступны в шаблоне в виде переменных. Компонент `FreeMarkerViewResolver` также позволяет экспортировать атрибуты запроса и сеанса в шаблоны FreeMarker. Для этого нужно установить свойство `exposeRequestAttributes` или `exposeSessionAttributes` (или оба) в значение `true`:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
      ↳ freemarker.FreeMarkerViewResolver">
  ...
  <property name="exposeRequestAttributes">
    <value>true</value>
  </property>
  <property name="exposeSessionAttributes">
    <value>true</value>
  </property>
</bean>
```

Здесь оба свойства установлены в `true`. В результате атрибуты и запроса, и сеанса будут копироваться в атрибуты шаблона и будут доступны в выражениях на языке шаблонов FreeMarker.

Связывание полей форм в шаблонах FreeMarker

Последнее, что осталось сделать, – связать поля формы со свойствами управляющего объекта. Выше вы уже видели, как для этой цели использовать теги JSP и макроопределения Velocity. Чтобы никого не обделить, фреймворк Spring предоставляет также множество макроопределений FreeMarker, повторяющих функциональность макроопределений Velocity, которые перечислены в табл. 8.3.

Таблица 8.3. Spring MVC предоставляет коллекцию макроопределений FreeMarker для связывания полей форм с объектом

Макроопределение	Назначение
<@spring.formCheckboxes path, options, separator, attributes />	Отображает набор флажков. Устанавливает состояние флажков согласно значениям свойств объекта
<@spring.formHiddenInput path, attributes />	Генерирует скрытое поле, связанное со свойством объекта
<@spring.formInput path, attributes, fieldType />	Отображает текстовое поле, связанное со свойством объекта
<@spring.formMultiSelect path, options, attributes />	Отображает список выбора, допускающий возможность выбора нескольких пунктов одновременно, связанный со свойством объекта
<@spring.formPasswordInput path, attributes />	Отображает поле ввода пароля, связанное со свойством объекта
<@spring.formRadioButtons path, options, separator, attributes />	Отображает набор радиокнопок, связанный со свойством объекта
<@spring.formSingleSelect path, options, attributes />	Отображает список выбора, допускающий возможность выбора единственного пункта, связанный со свойством объекта. Выбранное значение связано со свойством объекта
<@spring.formTextarea path, attributes />	Отображает текстовую область ввода, связанную со свойством объекта
<@spring.message messageCode />	Отображает «внешнее сообщение» из пакета ресурсов
<@spring.messageText messageCode, text />	Отображает «внешнее сообщение» из пакета ресурсов со значением по умолчанию, если сообщение не найдено в пакете ресурсов
<@spring.showErrors separator, class/style />	Отображает ошибки, выявленные при проверке
<@spring.url relativeUrl />	Отображает абсолютный URL по заданному относительному URL

За исключением незначительных синтаксических отличий, макроопределения FreeMarker идентичны макроопределениям Velocity. В листинге 8.18 представлен шаблон addRant.ftl представления addRant, демонстрирующий применение макроопределений.

Листинг 8.18. Шаблон FreeMarker представления для добавления сообщений

```
<#import "/spring.ftl" as spring /> <!-- Импортирование макроопределений -->
<html>                                <!-- Spring для FreeMarker -->
    <head>
        <title><@spring.message "title.addRant"/></title>
        <style>
            .error {
                color: #ff0000;
                font-weight: bold;
            }
        </style>
    </head>
    <body>
        <h2><@spring.message "title.addRant"/></h2>
        <form method="POST" action="addRant.htm">
            <!-- Связывание полей формы со свойствами объекта -->
            <b><@spring.message "field.state"/> </b><@spring.formInput
                "rant.vehicle.state", "" /><br/>
            <b><@spring.message "field.plateNumber"/></b>
            <@spring.formInput "rant.vehicle.plateNumber", "" /><br/>
            <@spring.message "field.rantText"/>
            <@spring.formTextarea "rant.rantText", "rows='5' cols='50'" />

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

Сравнив листинги 8.16 и 8.18, можно заметить поразительное их сходство. Однако между ними существуют два тонких отличия. Вместо `#springFormInput` в версии для FreeMarker используется `<@spring.formInput>`. А вместо `#springFormTextarea` используется `<@spring.formTextarea>`.

Кроме того, в отличие от Velocity, где макроопределения доступны автоматически, макроопределения FreeMarker необходимо импортировать. Первая строка в файле `addRant.ftl` импортирует макроопределения Spring для FreeMarker.

Как и при использовании механизма шаблонов Velocity, чтобы получить доступ к макроопределениям FreeMarker, их нужно подключить, установив свойство `exposeMacroHelpers` компонента `FreeMarkerViewResolver` в значение `true`:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.
      ↗ freemarker.FreeMarkerViewResolver">
  <property name="suffix" value=".ftl" />
  <property name="exposeSpringMacroHelpers" value="true" />
</bean>
```

Теперь у вас на выбор есть три механизма шаблонов, пригодных для создания веб-приложений в Spring. Но все эти механизмы воспроизводят разметку HTML, а иногда возникает потребность в воспроизведении документов в других форматах. Поэтому двинемся дальше и посмотрим, как генерировать документы в форматах, отличных от HTML.

8.7. Генерирование вывода, отличного от HTML

До сих пор веб-слой приложения производил вывод только в формате HTML. В действительности HTML – наиболее типичный способ отображения информации в веб. Но формат HTML не всегда соответствует представляемой информации.

Например, если данные представляются в табличной форме, эту информацию предпочтительнее было бы представлять в форме электронной таблицы. Электронные таблицы также могут пригодиться, когда требуется дать пользователям приложения возможность манипулировать представленными данными.

Возможно, вам потребуется более точно управлять форматированием документов. Точное форматирование HTML-документов практически невозможно, особенно когда они просматриваются в различных типах браузеров. Однако имеется формат PDF (Portable Document Format – переносимый формат документов), де-факто ставший стандартом для создания документов с точным форматированием, который может просматриваться на разных платформах.

Электронные таблицы и документы PDF – это обычные файлы. Но в Spring имеются классы представлений, позволяющие создавать электронные таблицы и документы PDF динамически, опираясь на данные, имеющиеся в приложении.

Давайте исследуем поддержку не-HTML-представлений в Spring, начав с динамического создания электронных таблиц в формате Excel.

8.7.1. Создание электронных таблиц Excel

Занимающиеся разработкой программного обеспечения достаточно-но продолжительное время могли заметить, что электронные таблицы Microsoft Excel прочно обосновались в бизнесе. Плохо ли, хорошо ли, но бизнесмены обожают электронные таблицы. С помощью электронных таблиц они обмениваются информацией, анализируют ее, составляют диаграммы, планируют и принимают решения. Используя Excel – и работа во многих компаниях застопорится.

Учитывая такую любовь к электронным таблицам на рабочих местах, очень важно иметь возможность генерировать электронные таблицы в приложении. И такая возможность есть: в состав Spring входит `AbstractExcelView` – представление, подходящее для создания электронных таблиц.

Данные, представляемые приложением RoadRantz, сложно назвать важными деловыми данными, поэтому нам остается только предполагать, что многие автолюбители, находящиеся в дороге, являются бизнесменами. И вполне возможно, что, управляя автомобилем, они продолжают думать о фактах и цифрах (и электронных таблицах). Поскольку такие автолюбители настолько тесно связаны с электронными таблицами, возможно, имеет смысл предоставить этим бизнесменам списки отзывов об их автомобилях в форме электронной таблицы.

Фреймворк Spring поддерживает возможность вывода электронных таблиц Excel посредством `AbstractExcelView`. Как следует из имени, это абстрактный класс, на основе которого нужно создать свой подкласс, где определить все особенности электронной таблицы.

В качестве примера создания подкласса абстрактного класса `AbstractExcelView` рассмотрим класс `RantExcelView` (листинг 8.19). Эта реализация представления воспроизводит список сообщений в формате электронной таблицы Excel.

Листинг 8.19. Список сообщений в виде электронной таблицы

```
package com.roaddrantz.mvc;
import java.util.Collection;
import java.util.Iterator;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.poi.hssf.usermodel.HSSFCCellStyle;
import org.apache.poi.hssf.usermodel.HSSFDataFormat;
```

```
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;
import org.springframework.web.servlet.view.document.AbstractExcelView;
import com.roadrantz.domain.Rant;
import com.roadrantz.domain.Vehicle;

public class RantExcelView extends AbstractExcelView {
    protected void buildExcelDocument(
        Map model, HSSFWorkbook workbook,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {

        Collection rants = (Collection) model.get("rants");
        Vehicle vehicle = (Vehicle) model.get("vehicle");

        HSSFSheet sheet = createSheet(workbook,
            vehicle.getPlateNumber());

        HSSFCellStyle cellStyle = workbook.createCellStyle();
        cellStyle.setDataFormat(                      // Установка формата даты
            HSSFDataFormat.getBuiltinFormat("m/d/yy h:mm"));

        int rowNum = 1;
        for (Iterator iter = rants.iterator(); iter.hasNext();) { // Добавление
            Rant rant = (Rant) iter.next();                         // сообщений
            rowNum = addRantRow(sheet, cellStyle, rowNum, rant);   // в таблицу
        }
    }

    private int addRantRow(HSSFSheet sheet, HSSFCellStyle cellStyle,
        int rowNum, Rant rant) {
        HSSFRow row = sheet.createRow(rowNum++);
        row.createCell((short) 0).setCellValue(rant.getPostedDate());
        row.createCell((short) 1).setCellValue(rant.getRantText());
        row.getCell((short) 1).setCellStyle(cellStyle);
        return rowNum;
    }

    private HSSFSheet createSheet(HSSFWorkbook workbook,
        String plateNumber) {
        HSSFSheet sheet = workbook.createSheet(
            "Rants for " + plateNumber);

        HSSFRow header = sheet.createRow(0);                      // Добавление строки
```

```
    header.createCell((short) 0).setCellValue("Date");      // заголовка
    header.createCell((short) 1).setCellValue("Text");
    return sheet;
}
}
```

В классе `RantExcelView` необходимо определить только один метод класса `AbstractExcelView` – `buildExcelDocument()`. Этот метод получает отображение (`Map`) с данными модели для электронной таблицы. Ему также передаются `HttpServletRequest` и `HttpServletResponse`, на тот случай, если потребуется дополнительная информация, отсутствующая в модели. Однако в данном примере используются только данные модели.

Кроме того, метод `buildExcelDocument()` получает `HSSFWorkbook` – компонент из `Jakarta POI`, представляющий рабочую книгу (`workbook`) `Excel`¹. Реализация собственного представления в `Spring MVC` для вывода файлов `Excel` – это лишь вопрос использования библиотеки `POI` для отображения модели данных в рабочую книгу. В `RantExcelView` список сообщений извлекается из модели данных и в цикле, построчно добавляется в электронную таблицу.

Чтобы сделать класс `RantExcelView` доступным в `Spring MVC`, его необходимо зарегистрировать с помощью `ResourceBundleViewResolver` или `XmlViewResolver`. Мы воспользуемся `XmlViewResolver`, поэтому следующего объявления в `roadrantz-views.xml` будет достаточно:

```
<bean id="vehicleRants.xls"
      class="com.roadrantz.mvc.RantExcelView" />
```

Теперь, чтобы продемонстрировать работу `RantExcelView`, необходимо создать контроллер, помещающий список сообщений в модель данных. Так получилось, что имеющийся контроллер `RantsForVehicleController` уже делает все необходимое. Единственная проблема в том, что `RantsForVehicleController` уже используется для отображения разметки `HTML`. Нам необходимо немного изменить его, чтобы придать дополнительную гибкость в выборе представления.

¹ Префикс «`HSSF`» в имени `HSSFWorkbook` и в именах других классов из `POI` является аббревиатурой от «`Horrible SpreadSheet Format`» (отвратительный формат электронных таблиц). Видимо, у разработчиков `POI` сформировалось устойчивое мнение о формате файлов `Excel`.

Первое, что нужно сделать, – добавить URL отображения запроса на получение книги Excel. Добавьте следующий элемент <prop> в свойство mappings компонента urlMapping:

```
<prop key="/rantsForVehicle.xls">
    rantsForVehicleController
</prop>
```

Если теперь DispatcherServlet примет запрос для /rantsForVehicle.xls, он передаст его компоненту rantsForVehicleController. Это тот же контроллер, что обслуживает запросы для /rantsForVehicle.htm. Но как он определит, какое представление следует использовать, Excel или JSP?

URI запроса содержит подсказку, описывающую тип представления. URI запроса HTML-страницы уже заканчивается расширением htm. Запросы на получение данных в формате Excel будут отличаться по расширению xls в конце URI. Следующий метод getViewName() извлекает расширение из URI и использует его при конструировании имени представления:

```
private static final String BASE_VIEW_NAME = "vehicleRants";
private String getViewName(HttpServletRequest request) {
    String requestUri = request.getRequestURI();
    String extension = "." +
        requestUri.substring(requestUri.lastIndexOf("."));
    if("htm".equals(extension)) { extension=""; }
    return BASE_VIEW_NAME + extension;
}
```

Если URI оканчивается расширением htm, тогда считается, что получен запрос на получение HTML-страницы и арбитру InternalResourceViewResolver будет позволено выбрать представление JSP. В противном случае именем представления будет vehicleRants, за которым следует расширение URI.

Далее необходимо изменить метод handle() контроллера RantsForVehicleController и задействовать в нем метод getViewName() при выборе представления:

```
protected ModelAndView handle(HttpServletRequest request,
    HttpServletResponse response, Object command,
    BindException errors) throws Exception {
    ...
    return new ModelAndView(getViewName(request), model);
}
```

Осталось решить еще одну проблему. Сервлет DispatcherServlet никогда не получит запроса для адреса /rantsForVehicle.xls, потому что он настроен в web.xml для обработки запросов *.htm. Поэтому необходимо настроить еще один элемент <servletmapping>, как показано ниже:

```
<servlet-mapping>
    <servlet-name>roadrantz</servlet-name>
    <url-pattern>*.xls</url-pattern>
</servlet-mapping>
```

Теперь DispatcherServlet будет обрабатывать оба типа запросов, *.htm и *.xls, и приложение будет способно генерировать списки сообщений в формате Excel.

Электронная таблица предназначается в первую очередь для тех, кто привык к работе с Excel. Но других они могут отпугивать. Для таких пользователей желательно выводить информацию в более дружественном формате. Чтобы осчастливить их, посмотрим, как с помощью Spring организовать вывод информации в виде документов PDF.

8.7.2. Создание документов PDF

Документы PDF часто используются в Интернете для отображения информации в формате, одинаково точном и универсальном. Несмотря на то что каскадные таблицы стилей (Cascading Style Sheets, CSS) достигли определенного совершенства, обеспечив профессиональный уровень форматирования документов HTML, они имеют некоторые ограничения. Форматирование содержимого документов PDF, напротив, практически не имеет ограничений.

Кроме того, реализации CSS в разных браузерах могут отличаться, тогда как документы PDF отображаются в Adobe Acrobat Viewer совершенно идентично на всех plataформах.

Допустим, что в добавок к отображению списка сообщений в формат электронной таблицы Excel нам необходимо реализовать вывод того же списка в формате PDF. Используя формат PDF, можно быть уверенными, что выходные данные будут отображаться одинаково у всех пользователей.

Поддержка отображения документов PDF в представлении реализована в Spring MVC в виде абстрактного класса AbstractPdfView. По аналогии с AbstractExcelView необходимо определить подкласс,

наследующий `AbstractPdfView`, и реализовать метод `buildPdfDocument()`, конструирующий документ PDF.

В листинге 8.20 приводится класс `RantPdfView`, наследующий класс `AbstractPdfView` и реализующий вывод списка сообщений в формате PDF.

Листинг 8.20. Список сообщений в формате PDF

```
package com.roadrantz.mvc;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.view.document.AbstractPdfView;
import com.lowagie.text.Document;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;
import com.roadrantz.domain.Rant;

public class RantPdfView extends AbstractPdfView {
    protected void buildPdfDocument(Map model, Document document,
        PdfWriter pdfWriter, HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        List rants = (List) model.get("rants");
        Table rantTable = new Table(4); // Создание таблицы
        rantTable.setWidth(90);
        rantTable.setBorderWidth(1);

        rantTable.addCell("State");           // Добавление строки заголовков
        rantTable.addCell("Plate");
        rantTable.addCell("Date Posted");
        rantTable.addCell("Text");

        for (Iterator iter = rants.iterator(); iter.hasNext();) {
            Rant rant = (Rant) iter.next();

            rantTable.addCell(rant.getVehicle().getState());           // Добавление
            rantTable.addCell(rant.getVehicle().getPlateNumber()); // каждого
            rantTable.addCell(rant.getPostedDate().toString());     // сообщения
            rantTable.addCell(rant.getRantText());
        }
        document.add(rantTable);
    }
}
```

Подобно методу `buildExcelDocument()` в классе `AbstractExcelView`, метод `buildPdfDocument()` получает данные модели в виде отображения (`Map`), а также `HttpServletRequest` и `HttpServletResponse`. Дополнительно ему передаются объекты классов `Document` и `PdfWriter`. Эти два класса являются частью библиотеки `iText PDF` (www.lowagie.com/iText) и используются для конструирования документов PDF. За дополнительной информацией о библиотеке `iText` я рекомендую обратиться к книге «*iText in Action*» (Manning, 2006).

В объекте `Document` методу `buildPdfDocument()` передается пустой документ `iText`, ожидающий наполнения содержимым. В `RantPdfView` список сообщений извлекается из модели, и на его основе конструируется таблица, где каждому сообщению отводится одна строка. После добавления в таблицу всех сообщений таблица добавляется в `Document`.

Чтобы обеспечить доступность представления `RantPdfView` для Spring MVC, добавим его объявление в файл `roadrantzviews.xml` рядом с объявлением `RantExcelView`:

```
<bean id="vehicleRants.pdf"
      class="com.roadrantz.mvc.RantPdfView" />
```

Теперь любой контроллер, возвращающий `ModelAndView` с именем представления `vehicleRants.pdf`, получит представление, отображаемое компонентом `RantPdfView`.

В разделе 8.6.1 мы уже реализовали в контроллере `RantsForVehicleController` динамический выбор представления на основе расширения URI. Поэтому никаких дальнейших изменений в `RantsForVehicleController` не требуется, чтобы обеспечить поддержку `RantPdfView`. Достаточно просто зарегистрировать отображение URL, чтобы `DispatcherServlet` смог передать запрос к адресу `/rantsForVehicle.pdf` контроллеру `RantsForVehicleController`:

```
<prop key="/rantsForVehicle.pdf">
    rantsForVehicleController
</prop>
```

Кроме того, по аналогии с представлением, реализующим вывод данных в формате Excel, необходимо создать `<servlet-mapping>` в `web.xml`, чтобы обеспечить передачу запросов `*.pdf` сервлету `DispatcherServlet`:

```
<servlet-mapping>
    <servlet-name>roadrantz</servlet-name>
    <url-pattern>*.pdf</url-pattern>
</servlet-mapping>
```

Классы `AbstractExcelView` и `AbstractPdfView`, входящие в состав фреймворка Spring, позволяют быстро реализовать вывод документов в форматах Excel и PDF.

8.8. В заключение

В этой главе была реализована большая часть веб-слоя приложения Spitter. Как было показано выше, в состав Spring входит мощный и гибкий веб-фреймворк. Аннотации из фреймворка Spring MVC обеспечивают возможность использования модели программирования, близкую к POJO, облегчая создание контроллеров, обрабатывающих запросы и простых в тестировании. Обычно контроллеры не участвуют в процессе обработки непосредственно, а делегируют эту работу другим компонентам в приложениях на основе Spring, которые внедряются в контроллеры с применением приема внедрения зависимостей.

Используя механизм отображения при выборе контроллера для обработки запроса и арбитров представлений, определяющих порядок вывода результатов, фреймворк Spring MVC обеспечивает слабую связность между тем, как будет обрабатываться запрос, и тем, как будут представлены результаты. Это делает Spring более привлекательным, по сравнению со многими другими веб-фреймворками MVC, где выбор ограничивается одним или двумя вариантами.

Несмотря на то что в этой главе представления были реализованы как JSP-страницы, обеспечивающие вывод разметки HTML, нет ничего, что помешало бы реализовать вывод модели данных в других форматах, включая XML и JSON. В главе 11 будет показано, как превратить веб-слой приложения Spitter в мощную веб-службу, где мы исследуем поддержку REST в Spring.

А сейчас продолжим знакомство с особенностями конструирования веб-приложений с помощью Spring и исследуем Spring Web Flow – расширение фреймворка Spring MVC, позволяющее разрабатывать диалоговые веб-приложения на основе Spring.



Глава 9. Использование Spring Web Flow

В этой главе рассматриваются следующие темы:

- ❑ создание диалоговых веб-приложений;
- ❑ определение состояний и операций этапов;
- ❑ безопасность диалоговых веб-приложений.

Одна из удивительных особенностей Интернета состоит в том, что в нем легко потеряться. В нем сосредоточено так много всего, что хотелось бы посмотреть и почитать. Основой широких возможностей Интернета являются гиперссылки. Нет ничего удивительного, что их также называют Всемирной паутиной. Как настоящая паутина, которую плетет паук, она способна поймать в ловушку любого, кто попытается пройтись по нему.

Честно признаюсь, что одна из причин, почему я так долго писал эту книгу, состоит в том, что я однажды просто заблудился в бесконечных ссылках в Википедии.

Бывают моменты, когда веб-приложение должно взять на себя управление перемещениями пользователя, ведя его за руку по страницам приложения. Наиболее наглядным примером такого поведения является процесс оформления покупки на сайте электронной коммерции. Начиная со списка товаров, приложение ведет пользователя через процесс ввода параметров доставки, оформления счета и, наконец, к подтверждению заказа.

Spring Web Flow – это веб-фреймворк, позволяющий разрабатывать элементы, представляющие этапы некоторого процесса. В этой главе мы займемся исследованием Spring Web Flow и посмотрим, какое место он занимает в иерархии Spring.

Приложение, выполняющее последовательность операций в определенном порядке, можно написать с помощью любого веб-фреймворка. Мне приходилось даже видеть приложения на основе фреймворка Struts, соблюдающие определенную очередность этапов выпол-

нения. Но, не имея возможности отделять определение последовательности операций от реализации, вы быстро обнаружите, что определение последовательности «размазывается» по различным элементам, составляющим процесс. В таких приложениях отсутствует какое-то одно, определенное место, взглянув на которое можно было бы получить полное представление о процессе.

Spring Web Flow является расширением фреймворка Spring MVC, позволяющим разрабатывать диалоговые веб-приложения, предусматривающие пошаговое выполнение операций, и отделяющим определение этапов процесса от классов и представлений, реализующих последовательность операций.

С целью знакомства с расширением Spring Web Flow мы оторвемся от примера приложения Spitter и создадим новое веб-приложение, обеспечивающее оформление заказа пиццы. Расширение Spring Web Flow будет использоваться для определения процедуры заказа.

Прежде чем приступать к работе с расширением Spring Web Flow, его необходимо установить в проект. С этого мы и начнем.

9.1. Установка Spring Web Flow

Несмотря на то что Spring Web Flow является частью проекта Spring Framework, это расширение не входит в состав Spring Framework непосредственно. То есть, прежде чем приступать к созданию диалоговых приложений, выполняющих операции в определенной последовательности, необходимо добавить расширение Spring Web Flow в библиотеку классов (classpath) приложения.

Загрузить расширение Spring Web Flow можно на домашней странице проекта (<http://www.springframework.org/webflow>). Выбирайте для загрузки самую свежую версию (на момент написания этих строк последней была версия 2.2.1). Загрузив и распаковав zip-файл дистрибутива, можно найти следующие JAR-файлы расширения Spring Web Flow:

- org.springframework.binding-2.2.1.RELEASE.jar;
- org.springframework.faces-2.2.1.RELEASE.jar;
- org.springframework.js-2.2.1.RELEASE.jar;
- org.springframework.js.resources-2.2.1.RELEASE.jar;
- org.springframework.webflow-2.2.1.RELEASE.jar.

Для реализации нашего примера нам потребуются только JAR-файлы binding и webflow. Остальные предназначены для использования Spring Web Flow совместно с JSF и JavaScript.



9.1.1. Настройка расширения *Web Flow* в *Spring*

Расширение Spring Web Flow реализовано на основе фреймворка Spring MVC. Это означает, что все запросы в последовательности операций сначала обрабатываются сервлетом `DispatcherServlet`. Затем в контексте приложения Spring следует настроить несколько специальных компонентов, реализующих обработку запросов и выполняющих последовательность операций.

Некоторые компоненты, являющиеся частью последовательности, объявляются с использованием конфигурационных элементов из пространства имен Spring Web Flow. Поэтому необходимо добавить объявление пространства имен в XML-файл определения контекста приложения:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:flow="http://www.springframework.org/schema/webflow-config"
       xsi:schemaLocation="http://www.springframework.org/schema/webflow-config
                           http://www.springframework.org/schema/webflow-config/
                           spring-webflow-config-2.0.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
```

После объявления пространства имен можно приступать к связыванию компонентов, начиная с компонента-исполнителя последовательности.

Связывание компонента-исполнителя

Как следует из его названия, *компонент-исполнитель последовательности* (`flow executor`) управляет выполнением последовательности операций. Когда пользователь приступает к выполнению некоторой последовательности, компонент-исполнитель создает и запускает экземпляр, представляющий последовательность операций, для этого пользователя. Когда выполнение последовательности приостанавливается (например, когда пользователю отправляется страница, подготовленная представлением), компонент-исполнитель возобновляет ее выполнение после совершения пользователем некоторого действия.

Компонент-исполнитель определяется в контексте Spring с помощью элемента `<flow:flow-executor>`:

```
<flow:flow-executor id="flowExecutor"
    flow-registry="flowRegistry" />
```

Компонент-исполнитель отвечает за создание и выполнение последовательностей операций, но он не отвечает за загрузку определений последовательностей. За загрузку отвечает другой компонент – реестр последовательности (*flow registry*), который мы создадим далее. В нашем примере мы будем ссылаться на компонент реестра последовательности по идентификатору `flowRegistry`¹.

Настройка реестра последовательности

Задача *реестра последовательности* состоит в том, чтобы загрузить определение последовательности операций и сделать его доступным для компонента-исполнителя. Настройка реестра последовательности в конфигурации Spring выполняется с помощью элемента `<flow:flow-registry>`, как показано ниже:

```
<flow:flow-registry id="flowRegistry"
    base-path="/WEB-INF/flows">
    <flow:flow-location-pattern value="*-flow.xml" />
</flow:flow-registry>
```

Согласно этому объявлению, компонент реестра будет искать определения последовательностей операций в каталоге `/WEB-INF/flows`, как указано в атрибуте `base-path`. Согласно элементу `<flow:flow-location-pattern>`, определениями последовательностей операций будут считаться все XML-файлы, имена которых оканчиваются на `-flow.xml`.

Каждая последовательность имеет свой идентификатор. При использовании элемента `<flow:flow-location-pattern>`, как в нашем примере, идентификатором последовательности операций будет служить путь к каталогу относительно `base-path`, или часть пути, которую представляет двойная звездочка. На рис. 9.1 показано, как определяется идентификатор последовательности операций.

При желании можно опустить атрибут `base-path` и явно указать местоположение файла с определением последовательности:

¹ В примере ниже атрибут `id` элемента `flow-registry` определяется явно, но в этом нет необходимости. Если опустить этот атрибут, он получит значение по умолчанию `flowRegistry`.



Рис. 9.1. При использовании элемента `<flow:flow-location-pattern>` путь к файлу с определением последовательности, относительно указанного каталога, будет использоваться как идентификатор последовательности

```
<flow:flow-registry id="flowRegistry">
    <flow:flow-location path="/WEB-INF/flows/springpizza.xml" />
</flow:flow-registry>
```

Здесь вместо элемента `<flow:flow-locationpattern>` использован элемент `<flow:flow-location>`. Атрибут `path` непосредственно указывает на файл `/WEB-INF/flows/springpizza.xml` с определением последовательности операций. Когда настройка выполняется таким способом, идентификатором последовательности становится базовое имя файла определения – в данном случае `springpizza`.

Если потребуется еще более явно определить идентификатор последовательности, его можно указать в атрибуте `id` элемента `<flow:flow-location>`. Например, следующий элемент `<flow:flow-location>` определяет идентификатор `pizza` последовательности:

```
<flow:flow-registry id="flowRegistry">
    <flow:flow-location id="pizza"
        path="/WEB-INF/flows/springpizza.xml" />
</flow:flow-registry>
```

Обработка запросов к последовательности

Как было показано в предыдущей главе, контроллер `DispatcherServlet` обычно передает запросы другим контроллерам для обработки. Но в случае с последовательностями операций в помощь контроллеру `DispatcherServlet` необходимо создать компонент `FlowHandlerMapping`, чтобы обеспечить передачу запросов расширению `Spring Web Flow`. Настройка компонента `FlowHandlerMapping` в контексте приложения `Spring` выполняется, как показано ниже:

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
    <property name="flowRegistry" ref="flowRegistry" />
</bean>
```

Как показано в этом фрагменте, в компонент FlowHandlerMapping внедряется ссылка на реестр последовательности, чтобы он смог отображать URL-адреса запросов на последовательность. Например, для последовательности с идентификатором pizza компонент FlowHandlerMapping будет знать, что запросы с URL /pizza (относительно пути контекста приложения) должны отображаться в эту последовательность.

Компонент FlowHandlerMapping отвечает за направление запросов к последовательности в Spring Web Flow, а за их обработку отвечает компонент FlowHandlerAdapter. Компонент FlowHandlerAdapter является эквивалентом контроллера Spring MVC, в том смысле что он обрабатывает входящие запросы к последовательности. Компонент FlowHandlerAdapter определяется, как показано ниже:

```
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
    <property name="flowExecutor" ref="flowExecutor" />
</bean>
```

Этот компонент-обработчик играет роль моста между DispatcherServlet и Spring Web Flow. Он обрабатывает запросы к последовательности и управляет ими. Здесь в него внедряется ссылка на компонент-исполнитель, управляющий последовательность, запросы к которой обрабатывает данный компонент.

Мы настроили все компоненты, необходимые для работы Spring Web Flow. Осталось лишь определить саму последовательность операций. Это будет сделано чуть ниже, но сначала познакомимся с элементами, составляющими последовательность.

9.2. Элементы последовательности операций

В Spring Web Flow последовательность операций (flow) определяется тремя основными элементами: состояниями, переходами и данными последовательности.

Состояния – это точки в последовательности, где что-то происходит. Если представить последовательность как поездку по дороге, тогда состояниями в этой поездке будут города, места стоянки автомобилей и просто остановки по пути. В отличие от поездки, при достижении некоторого состояния в последовательности, вместе того чтобы вынуть сумку с чипсами и кока-колой, приложение



выполняет некоторые действия, принимает какие-то решения или отправляет пользователю некоторую страницу.

Если состояния в последовательности можно сравнить с точками на карте, где выполняются остановки, то *переходы* – это дороги, соединяющие эти точки. Выполняя последовательность, приложение перемещается от одного состояния к другому посредством переходов.

Путешествуя от города к городу, вы покупаете какие-нибудь сувениры, памятные предметы и опустошаете пакеты с едой. Аналогично, по мере выполнения последовательности, собираются какие-то данные, характеризующие условия выполнения. Меня так и тянет назвать эти данные состоянием последовательности, но слово *состояние* уже занято и при обсуждении имеет иной смысл.

Посмотрим, как эти три элемента определяются в Spring Web Flow.

9.2.1. Состояния

Фреймворк Spring Web Flow определяет пять различных типов состояний, которые перечислены в табл. 9.1.

Набор состояний, предоставляемых фреймворком Spring Web Flow, позволяет конструировать диалоговые веб-приложения практически любой сложности. Разумеется, далеко не все последовательности требуют использования всех состояний, описанных в табл. 9.1, но рано или поздно вам наверняка придется использовать их все.

Таблица 9.1. Набор состояний, определяемых фреймворком Spring Web Flow

Состояние	Описание
Действие	Действие – это место в последовательности, где выполняется некоторая логика
Решение	Решение – это точка ветвления последовательности на два направления, исходя из накопленных данных
Конец	Конец – это последняя остановка в последовательности. По достижении этого состояния выполнение последовательности завершается
Подпоследовательность	В состоянии «подпоследовательность» запускается новая последовательность, выполняющаяся в контексте текущей последовательности
Представление	Представление вызывает приостановку последовательности и дает пользователю возможность принять участие в процессе выполнения

Ниже будет показано, как из этих состояний формируется полная последовательность. Но сначала познакомимся, как каждый из этих элементов последовательности объявляется в определении последовательности.

Состояния-представления

Состояния-представления используются для вывода информации и дают пользователю возможность принять активное участие в выполнении последовательности. Фактически роль состояния-представления может играть любое представление, поддерживаемое фреймворком Spring MVC, но чаще всего они реализуются в виде страниц JSP.

Для объявления состояния-представления в XML-файле с определением последовательности используется элемент `<view-state>`:

```
<view-state id="welcome" />
```

В этом простом примере атрибут `id` играет двойную роль. Во-первых, он идентифицирует состояние внутри последовательности. А во-вторых, поскольку здесь отсутствует явная ссылка на представление, определяет логическое имя представления `welcome`, которое должно отображаться при достижении данного состояния.

При необходимости с помощью атрибута `view` можно явно указать логическое имя используемого представления:

```
<view-state id="welcome" view="greeting" />
```

Если представление является формой, в атрибуте `model` можно указать объект, который должен быть связан с формой:

```
<view-state id="takePayment" model="flowScope.paymentDetails"/>
```

Здесь определено, что форма в представлении `takePayment` должна быть связана с объектом `paymentDetails`, в области видимости данной последовательности. (Подробнее об областях видимости последовательностей и данных рассказывается ниже.)

Состояния-действия

Состояния-представления вовлекают в работу последовательности пользователя, а состояния-действия сами выполняют некоторую работу. Состояния-действия обычно вызывают некоторые методы



компонентов, управляемых фреймворком Spring, и затем выполняют переход к другому состоянию, в зависимости от результатов, возвращаемых методом.

В определении последовательности состояния-действия объявляются с помощью элемента `<action-state>`. Например:

```
<action-state id="saveOrder">
    <evaluate expression="pizzaFlowActions.saveOrder(order)" />
    <transition to="thankYou" />
</action-state>
```

Хотя это и не обязательно, но элементы `<action-state>` часто включают вложенный элемент `<evaluate>`. Этот элемент дает состоянию-действию возможность выполнить некоторую работу. Атрибут `expression` определяет выражение, которое должно быть вычислено при входе в состояние. В данном случае в атрибуте `expression` указано выражение на языке SpEL¹, которое должно вызвать метод `saveOrder()` компонента с идентификатором `pizzaFlowActions`.

Состояния-решения

Последовательности могут быть исключительно линейными, выполняющими переход от одного состояния к другому, не предусматривающими альтернативных маршрутов. Но чаще в той или иной точке бывает необходимо выполнить ветвление последовательности в зависимости от сложившихся условий.

Состояния-решения позволяют продолжить выполнение последовательности в одном из двух направлений. Состояние-решение вычисляет логическое выражение и, исходя из результата, выбирает один из двух переходов. В определении последовательности состояния-решения объявляются с помощью элемента `<decision-state>`. Ниже представлен типичный пример объявления состояния-решения:

```
<decision-state id="checkDeliveryArea">
    <if test="pizzaFlowActions.checkDeliveryArea(customer.zipCode)"
        then="addCustomer"
        else="deliveryWarning" />
</decision-state>
```

¹ Начиная с версии 2.1.0, фреймворк Spring Web Flow использует язык выражений Spring Expression Language, но при желании можно также использовать OGNL или Unified EL.

Как видите, элемент `<decision-state>` действует не в одиночку. Основой состояния-решения является элемент `<if>`. Именно в нем вычисляется выражение. Если выражение вернет `true`, последовательность выполнит переход к состоянию, определяемому атрибутом `then`. В противном случае будет выполнен переход к состоянию, определяемому атрибутом `else`.

Состояния-подпоследовательности

Едва ли кому-то приходилось помещать всю логику работы приложения в один метод. Обычно приложение разбивается на множество классов, методов и других структур.

Точно так же бывает полезно разбить последовательность на дискретные части. Элемент `<subflow-state>` позволяет вызвать другую последовательность из текущей последовательности. Это напоминает вызов метода из другого метода.

Ниже приводится пример объявления элемента `<subflow-state>`:

```
<subflow-state id="order" subflow="pizza/order">
    <input name="order" value="order"/>
    <transition on="orderCreated" to="payment" />
</subflow-state>
```

Здесь элемент `<input>` используется для передачи объекта заказа в подпоследовательность. И, если подпоследовательность завершится в конечном состоянии `<end-state>` с идентификатором `orderCreated`, текущая последовательность выполнит переход к состоянию с идентификатором `payment`.

Но не будем забегать вперед. Мы еще не знакомы с элементом `<end-state>` и с переходами. О переходах рассказывается в разделе 9.2.2. А что касается конечных состояний, рассмотрим их прямо сейчас.

Конечные состояния

Рано или поздно выполнение любой последовательности должно завершиться. Именно это и случится по достижении конечного состояния. Элемент `<end-state>` обозначает конец последовательности и обычно объявляется, как показано ниже:

```
<end-state id="customerReady" />
```

Когда последовательность достигает элемента `<end-state>`, ее выполнение завершается. Что происходит дальше, зависит от нескольких факторов.



- ❑ Если завершившаяся последовательность является подпоследовательностью, управление вернется вызвавшей последовательности, которая продолжит выполнение с элемента `<subflow-state>`. Значение атрибута `id` элемента `<end-state>` будет использоваться как имя события для перехода из состояния `<subflow-state>`.
- ❑ Если элемент `<end-state>` имеет атрибут `view`, будет отображено указанное в нем представление. Имя представления в атрибуте `view` может определять путь к шаблону представления относительно последовательности, предваряясь приставкой `externalRedirect:` для ссылки на внешнюю для последовательности страницу или приставкой `flowRedirect:` для перехода в другую последовательность.
- ❑ Если завершившаяся последовательность не является подпоследовательностью и атрибут `view` в элементе `<end-state>` не определен, тогда последовательность просто завершит выполнение. Браузер остановится на базовом URL последовательности, при повторном обращении к которому (в отсутствие активной последовательности) будет запущен новый экземпляр последовательности.

Важно понимать, что последовательность может иметь несколько конечных состояний. Поскольку идентификатор конечного состояния определяет имя события, возбуждаемого в элементе `<subflow-state>`, иногда бывает желательно обеспечить возможность завершения последовательности в разных точках, чтобы возбудить разные события в вызывающей последовательности. Даже последовательности, которые никогда не будут вызываться как подпоследовательности, могут иметь несколько конечных состояний, отображающих различные конечные страницы, в зависимости от условий, сложившихся к моменту завершения.

Теперь, после знакомства с различными состояниями последовательностей, необходимо выяснить, как выполняются переходы между состояниями. Посмотрим, как прокладываются дороги в последовательностях.

9.2.2. Переходы

Как уже упоминалось выше, переходы связывают состояния внутренней последовательности. Каждое состояние в последовательности, за исключением конечных состояний, должно определять по мень-

шей мере один переход, чтобы последовательность знала, к какому состоянию следует перейти по завершении текущего. Состояние может определять несколько переходов, каждый из которых определяет свой путь выхода из текущего состояния.

Переходы определяются элементами `<transition>`, вложенными в элементы состояний (`<action-state>`, `<view-state>` и `<subflow-state>`). В простейшем случае элемент `<transition>` определяет следующее состояние в последовательности:

```
<transition to="customerReady" />
```

Атрибут `to` используется для ссылки на следующее состояние. Когда элемент `<transition>` объявляется с единственным атрибутом `to`, переход играет роль перехода по умолчанию к указанному состоянию и будет выполнен, если остальные переходы недопустимы.

Чаще переходы определяются как направление движения при определенном событии. В состоянии-представлении событием обычно является некоторое действие, выполненное пользователем. В состоянии-действии событие является результатом вычисления выражения. В случае с состоянием-подследовательностью событие определяется идентификатором конечного состояния, достигнутого подследовательностью. В любом случае, событие, определяющее направление перехода, можно объявить в атрибуте `on`:

```
<transition on="phoneEntered" to="lookupCustomer"/>
```

В этом примере последовательность выполнит переход к состоянию с идентификатором `lookupCustomer`, если будет возбуждено событие `phoneEntered`.

Последовательности могут также выполнять переход к другим состояниям в ответ на какое-либо исключение. Например, если запись с информацией о заказчике не будет найдена, можно выполнить переход к состоянию-представлению с регистрационной формой. Следующий фрагмент демонстрирует определение такого перехода:

```
<transition  
    on-exception=  
        "com.springinaction.pizza.service.CustomerNotFoundException"  
    to="registrationForm" />
```



Атрибут `onexception` близко напоминает атрибут `on`, за исключением того, что он определяет не событие, а исключение, вызывающее переход. В данном случае исключение `CustomerNotFoundException` вызовет переход к состоянию `registrationForm`.

Глобальные переходы

Иногда после создания последовательности обнаруживается, что одни и те же переходы определяются в нескольких состояниях. Например, я ничуть не удивился бы, обнаружив следующий элемент `<transition>`, встречающийся в последовательности несколько раз:

```
<transition on="cancel" to="endState" />
```

Вместо того чтобы вновь и вновь повторять одинаковые объявления переходов в нескольких состояниях, их можно объявить глобальными переходами, вложив элемент `<transition>` в элемент `<global-transitions>`. Например:

```
<global-transitions>
    <transition on="cancel" to="endState" />
</global-transitions>
```

При наличии такого определения все состояния в последовательности автоматически получат неявный переход `cancel`.

Итак, мы познакомились с состояниями и переходами. Но, прежде чем приступить к определению последовательностей, необходимо еще поговорить о данных в последовательностях – последнем элементе триады веб-последовательностей.

9.2.3. Данные в последовательностях

Если вам когда-либо приходилось играть в приключенческие игры, вы знаете, что при перемещении из одного пункта в другой персонажу иногда попадаются различные объекты, которые можно подобрать и забрать с собой. Иногда найденный объект можно сразу пустить в ход. Иногда его приходится нести с собой через всю игру, не представляя, для чего он может пригодиться, пока вы не доберетесь до заключительной головоломки и не обнаружите, что объект можно использовать для ее решения.

Последовательности очень похожи на такие игры. В процессе перемещения последовательности от состояния к состоянию она

приобретает различные данные. Иногда эти данные оказываются необходимы немедленно (возможно, только чтобы отобразить их на странице). Иногда их приходится хранить на протяжении всего времени выполнения последовательности и использовать только по завершении последовательности.

Объявление переменных

Данные в последовательностях хранятся в переменных, доступных из любой точки внутри последовательности. Они могут создаваться разными способами. Простейший из них заключается в том, чтобы объявить переменную внутри последовательности с помощью элемента `<var>`:

```
<var name="customer" class="com.springinaction.pizza.domain.Customer"/>
```

Здесь создается новый экземпляр объекта `Customer` и помещается в переменную с именем `customer`. Эта переменная будет доступна во всех состояниях внутри последовательности.

Переменную можно также создать с помощью элемента `<evaluate>` как часть состояния-действия или состояния-представления. Например:

```
<evaluate result="viewScope.toppingsList"
          expression="T(com.springinaction.pizza.domain.Topping).asList()" />
```

В данном случае элемент `<evaluate>` вычисляет выражение (на языке SpEL) и помещает результат в переменную с именем `toppingsList`, доступную только в области видимости представления. (Подробнее об областях видимости рассказывается чуть ниже.)

Аналогично значение переменной можно присвоить с помощью элемента `<set>`:

```
<set name="flowScope.pizza"
      value="new com.springinaction.pizza.domain.Pizza()" />
```

Элемент `<set>` действует практически так же, как элемент `<evaluate>`, присваивая результат выражения переменной. В данном случае переменной `pizza`, доступной в области видимости последовательности, присваивается ссылка на новый экземпляр объекта `Pizza`.

В разделе 9.3, где описывается пример определения последовательности, вы познакомитесь с дополнительными особенностя-



ми использования этих элементов. Но перед этим посмотрим, что означают для переменных выражения «в области видимости представления», «в области видимости последовательности» и другие «в области видимости...».

Области видимости данных в последовательностях

Данные, приобретаемые внутри последовательности, будут иметь различные области видимости в зависимости от контекста их использования. Фреймворк Spring Web Flow определяет пять областей видимости, перечисленные в табл. 9.2.

Таблица 9.2. Области видимости переменных в последовательностях Spring Web Flow

Область видимости	Описание
Диалог	Создается при запуске последовательности верхнего уровня и разрушается по ее завершении. Совместно используется последовательностью верхнего уровня и всеми ее подпоследовательностями
Последовательность	Создается при запуске последовательности верхнего уровня и разрушается по ее завершении. Доступна только внутри создавшей ее последовательности
Запрос	Создается, когда запрос превращается в последовательность, и разрушается по окончании его обработки
Кадр	Создается при запуске последовательности верхнего уровня и разрушается по ее завершении. Также разрушается после достижения состояния-представления
Представление	Создается при входе в состояние-представление и разрушается после выхода из него. Доступно только внутри состояния-представления

Переменные, объявляемые с помощью элемента `<var>`, всегда получают область видимости последовательности и доступны только внутри последовательности, где они объявлены. При использовании элементов `<set>` и `<evaluate>` область видимости определяется префиксом в значении атрибута `name` или `result`. Например, ниже показано, как присвоить значение переменной с именем `theAnswer`, доступной в области видимости потока:

```
<set name="flowScope.theAnswer" value="42"/>
```

Теперь, когда мы познакомились с основами веб-последовательностей, пришло время использовать полученные знания для соз-

дания полноценной веб-последовательности. В процессе работы подмечайте, как будет реализовано хранение данных в переменных с различными областями видимости.

9.3. Соединяем все вместе: последовательность pizza

Как уже говорилось выше в этой главе, мы оторвемся от примера приложения Spitter и создадим веб-приложение, с помощью которого голодные пользователи смогут заказать свою любимую итальянскую пиццу через Интернет¹.

Как оказывается, процедура заказа пиццы естественным образом укладывается в концепцию последовательностей. Для начала мы создадим последовательность верхнего уровня, определяющую всю процедуру заказа пиццы, а затем добавим несколько подпоследовательностей, определяющих операции более низкого уровня.

9.3.1. Определение основной последовательности

Руководство новой сети пиццерий Spizza² приняло решение, чтобы уменьшить нагрузку на отдел заказа пиццы по телефону, предоставить своим клиентам возможность заказать пиццу через Интернет. Когда клиент приходит на веб-сайт Spizza, он идентифицирует себя, выбирает одну или несколько пицц, добавляет их в заказ, предоставляет информацию об оплате, отправляет заказ и ждет доставки свежей и горячей пиццы. Эта последовательность операций представлена на рис. 9.2.

Прямоугольники на диаграмме представляют состояния, а стрелки – переходы. Как видите, процедура заказа пиццы представляет собой простую линейную последовательность операций. Этую процедуру легко можно реализовать с применением Spring Web Flow.

¹ Честно признаться, я не смог придумать, где в приложении Spitter можно было бы применить последовательность. Поэтому, вместо того чтобы насилию пытаться впихнуть пример использования Spring Web Flow в приложение Spitter, мы рассмотрим применение этого фреймворка на примере заказа пиццы.

² Да, я знаю, что в Сингапуре действительно существует пиццерия Spizza. Но речь совсем не о ней.

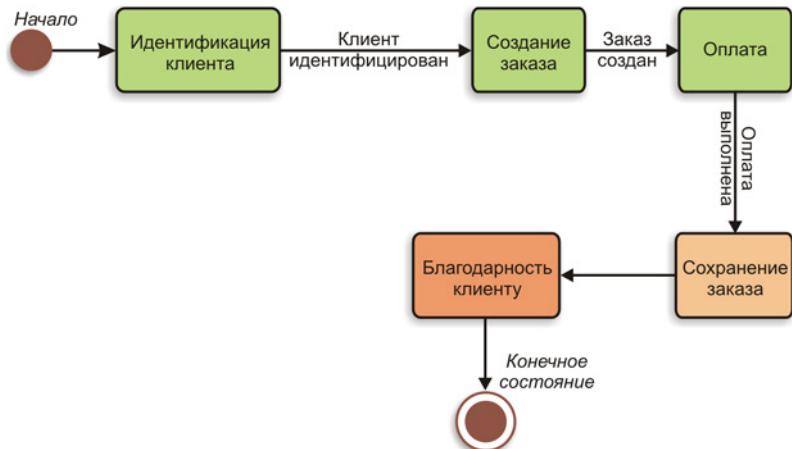


Рис. 9.2. Процедура заказа пиццы легко реализуется с помощью простой последовательности

Наибольший интерес здесь представляют первые три состояния, наиболее сложные в реализации, чем можно было бы заключить, глядя на простые прямоугольники.

В листинге 9.1 показано определение последовательности верхнего уровня, реализующей процедуру заказа пиццы.

Листинг 9.1. Реализация процедуры заказа пиццы в виде последовательности Spring Web Flow

```

<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                          http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <var name="order"
         class="com.springinaction.pizza.domain.Order"/>

    <!-- Вызов подпоследовательности идентификации клиента -->
    <subflow-state id="identifyCustomer" subflow="pizza/customer">
        <output name="customer" value="order.customer"/>
        <transition on="customerReady" to="buildOrder" />
    </subflow-state>

    <!-- Вызов подпоследовательности оформления заказа -->
  
```

```
<subflow-state id="buildOrder" subflow="pizza/order">
    <input name="order" value="order"/>
    <transition on="orderCreated" to="takePayment" />
</subflow-state>

<!-- Вызов подпоследовательности приема оплаты -->
<subflow-state id="takePayment" subflow="pizza/payment">
    <input name="order" value="order"/>
    <transition on="paymentTaken" to="saveOrder"/>
</subflow-state>

<action-state id="saveOrder">      <!-- Сохранить заказ -->
    <evaluate expression="pizzaFlowActions.saveOrder(order)" />
    <transition to="thankCustomer" />
</action-state>

<view-state id="thankCustomer">      <!-- Выражение благодарности клиенту -->
    <transition to="endState" />
</view-state>

<end-state id="endState" />

<global-transitions>      <!-- Глобальный переход в случае отмены заказа -->
    <transition on="cancel" to="endState" />
</global-transitions>
</flow>
```

Самое первое, что встречается в этом определении последовательности, – объявление переменной `order`. Каждый раз, когда будет запускаться эта последовательность, будет создаваться новый объект `Order`. Класс `Order`, представленный в листинге 9.2, имеет свойства, хранящие всю необходимую информацию о заказе, включая информацию о клиенте, список заказанных пицц и сведения об оплате.

Листинг 9.2. Класс `Order` хранит всю информацию, имеющую отношение к заказу

```
package com.springinaction.pizza.domain;

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

public class Order implements Serializable {
```

```
private static final long serialVersionUID = 1L;

private Customer customer;
private List<Pizza> pizzas;
private Payment payment;

public Order() {
    pizzas = new ArrayList<Pizza>();
    customer = new Customer();
}

public Customer getCustomer() {
    return customer;
}

public void setCustomer(Customer customer) {
    this.customer = customer;
}

public List<Pizza> getPizzas() {
    return pizzas;
}

public void setPizzas(List<Pizza> pizzas) {
    this.pizzas = pizzas;
}

public void addPizza(Pizza pizza) {
    pizzas.add(pizza);
}

public float getTotal() {
    return 0.0f;
}

public Payment getPayment() {
    return payment;
}

public void setPayment(Payment payment) {
    this.payment = payment;
}
```

Основной частью определения последовательности являются объявления состояний. По умолчанию первое состояние в определении последовательности является также состоянием, где произойдет первая остановка. В данном случае это состояние `identifyCustomer` (вызов подпоследовательности). Однако в случае необходимости можно явно определить, какое состояние будет первым, указав его идентификатор в атрибуте `start-state` элемента `<flow>`:

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                           http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd"
      start-state="identifyCustomer">
    ...
</flow>
```

Идентификация клиента, оформление заказа и получение оплаты – все эти операции слишком сложные, чтобы их можно было выразить в виде единственных состояний. Именно поэтому позднее мы определим их как самостоятельные последовательности. Но в последовательности верхнего уровня достаточно объявить их с помощью элемента `<subflow-state>`.

Переменная `order` будет заполняться в первых трех состояниях и затем сохраняться в четвертом. Для заполнения свойства `customer` объекта заказа состояние-подпоследовательность `identifyCustomer` использует элемент `<output>`, присваивая ему результат вызова подпоследовательности идентификации клиента. Состояния `buildOrder` и `takePayment` действуют иначе, передавая с помощью элемента `<input>` переменную `order` на вход своих подпоследовательностей, которые заполняют соответствующие свойства заказа внутри.

После заполнения заказа информацией о клиенте, количестве пицц и об оплате заказ можно сохранить. Эту задачу выполняет состояние `saveOrder`, являющееся состоянием-действием. В нем используется элемент `<evaluate>`, вызывающий метод `saveOrder()` компонента с идентификатором `pizzaFlowActions`, которому передается сохраняемый заказ. После сохранения заказа это состояние выполняет переход к состоянию `thankCustomer`.

Состояние `thankCustomer` – это простое состояние-представление, возвращающее JSP-файл `/WEB-INF/flows/pizza/thankCustomer.jsp`, представленный в листинге 9.3.



Листинг 9.3. Представление в формате JSP, отображающее благодарность клиенту за сделанный заказ

```
<html xmlns:jsp="http://java.sun.com/JSP/Page">
    <jsp:output omit-xml-declaration="yes"/>
    <jsp:directive.page contentType="text/html;charset=UTF-8" />

    <head><title>Spizza</title></head>

    <body>
        <h2>Thank you for your order!</h2>
        <!-- Возбуждает заключительное событие -->
        <![CDATA[
            <a href='${flowExecutionUrl}&_eventId=finished'>Finish</a>
        ]]>
    </body>
</html>
```

Эта страница выражает благодарность клиенту за заказ и содержит ссылку, позволяющую клиенту завершить выполнение последовательности. Эта ссылка является наиболее интересным элементом страницы, потому что является единственным пунктом, где пользователь имеет возможность взаимодействовать с последовательностью.

Фреймворк Spring Web Flow предоставляет переменную `flowExecutionUrl`, содержащую URL последовательности, для использования в представлении. Ссылка **Finish** (Конец) включает в URL параметр `_eventId`, обеспечивающий передачу события `finished` обратно в последовательность. Это событие переводит последовательность в конечное состояние.

В конечном состоянии последовательность прекращает выполнение. Поскольку в конечном состоянии не указано, куда выполнить переход по завершении последовательности, она продолжит работу сначала, с состояния `identifyCustomer`, готовая принять следующий заказ.

Это все, что относится к последовательности заказа пиццы верхнего уровня. Но нам необходимо определить еще подпоследовательности для состояний `identifyCustomer`, `buildOrder` и `takePayment`. Определим их далее, начав с подпоследовательности идентификации клиента.

9.3.2. Сбор информации о клиенте

Если прежде вам доводилось заказывать пиццу, вы наверняка знакомы с этой процедурой. Сначала у вас спрашивают номер телефона. Кроме возможности позвонить вам, если разносчик пиццы не

сможет отыскать ваш дом, номер телефона для пиццерии служит дополнительным идентификационным признаком. Если вы являетесь постоянным клиентом, в пиццерии могут использовать номер телефона для определения вашего адреса, чтобы узнать, куда доставлять заказ.

Если вы – новый клиент, номер телефона не позволит получить дополнительную информацию о вас. Поэтому следующее, что у вас спросят, – ваш адрес. На этом этапе в пиццерии уже будут знать, кто вы и куда доставить вашу пиццу. Но, прежде чем у вас спросят, какую пиццу вы хотели бы заказать, они проверят, попадает ли ваш адрес в зону их обслуживания. Если вы находитесь вне пределов досягаемости, вам придется самому прийти за готовой пиццей.

Порядок следования первого и всех последующих вопросов, которые задаются при заказе любой пиццы, можно схематически представить, как показано на рис. 9.3.

Эта последовательность гораздо интереснее последовательности верхнего уровня. Она нелинейна и имеет пару ветвлений в зависи-

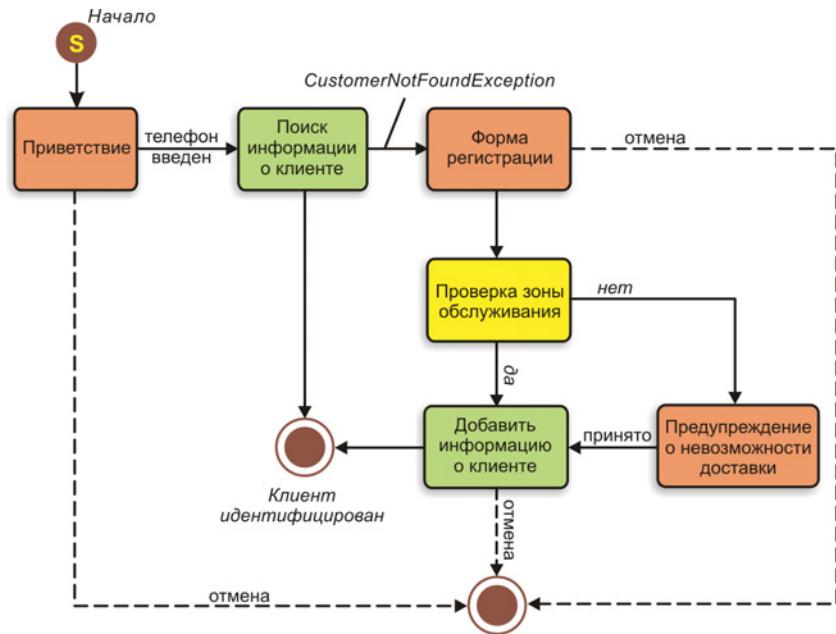


Рис. 9.3. Последовательность идентификации клиента получается более закрученной, чем основная последовательность



мости от сложившихся условий. Например, после попытки отыскать информацию о клиенте последовательность может завершиться (если информация найдена) или перейти к форме регистрации (если информация не найдена). Кроме того, в состоянии checkDeliveryArea может потребоваться, а может не потребоваться предупредить клиента, что он находится вне зоны обслуживания.

В листинге 9.4 представлено определение последовательности, выполняющей идентификацию клиента.

Листинг 9.4. Реализация идентификации голодного клиента в виде веб-последовательности

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                           http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <var name="customer" class="com.springinaction.pizza.domain.Customer"/>

    <view-state id="welcome">                                <!-- Вывод приветствия -->
        <transition on="phoneEntered" to="lookupCustomer"/>
    </view-state>

    <action-state id="lookupCustomer">                         <!-- Поиск информации о клиенте -->
        <evaluate result="customer" expression=
                  "pizzaFlowActions.lookupCustomer(requestParameters.phoneNumber)" />
        <transition to="registrationForm" on-exception=
                  "com.springinaction.pizza.service.CustomerNotFoundException" />
        <transition to="customerReady" />
    </action-state>

    <!-- Регистрация нового клиента -->
    <view-state id="registrationForm" model="customer">
        <on-entry>
            <evaluate expression=
                      "customer.phoneNumber = requestParameters.phoneNumber" />
        </on-entry>
        <transition on="submit" to="checkDeliveryArea" />
    </view-state>

    <decision-state id="checkDeliveryArea">                     <!-- Проверка зоны обслуживания -->
        <if test="pizzaFlowActions.checkDeliveryArea(customer.zipCode)"
            then="addCustomer"
```

```
else="deliveryWarning"/>
</decision-state>

<view-state id="deliveryWarning">    <!-- Вывод предупреждения -->
    <transition on="accept" to="addCustomer" />
</view-state>

<action-state id="addCustomer">      <!-- Добавление информации о клиенте -->
    <evaluate expression="pizzaFlowActions.addCustomer(customer)" />
    <transition to="customerReady" />
</action-state>

<end-state id="cancel" />

<end-state id="customerReady">
    <output name="customer" />
</end-state>

<global-transitions>
    <transition on="cancel" to="cancel" />
</global-transitions>
</flow>
```

В этом определении последовательности присутствуют ранее не использовавшиеся элементы, включая элемент `<decision-state>`. Кроме того, поскольку эта последовательность является подпоследовательностью, вызываемой из основной последовательности, предполагается, что ей будет передаваться объект `Order`.

Как и прежде, рассмотрим определение последовательности, состояние за состоянием, начав с состояния `welcome`.

Запрос номера телефона

Состояние `welcome` – это очень простое состояние-представление, приветствующее клиента, пришедшего на веб-сайт Spizza, и предлагающее ввести номер телефона. Само состояние не содержит ничего интересного. В нем определены два перехода: один ведет к состоянию `lookupCustomer`, если представление возбудит событие `phoneEntered`, и второй – глобальный переход `cancel`, который выполняется в ответ на событие `cancel`.

Наибольший интерес в состоянии `welcome` представляет само представление. Оно определено в файле `/WEB-INF/flows/pizza/customer/welcome.jspx`, как показано в листинге 9.5.

Листинг 9.5. Представление, приветствующее клиента и предлагающее ввести номер телефона

```
<html xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:form="http://www.springframework.org/tags/form">
<jsp:output omit-xml-declaration="yes"/>
<jsp:directive.page contentType="text/html;charset=UTF-8" />

<head><title>Spizza</title></head>

<body>
    <h2>Welcome to Spizza!!!</h2>

    <form:form>
        <input type="hidden" name="_flowExecutionKey"
               value="${flowExecutionKey}" />

        <input type="text" name="phoneNumber"/><br/>

        <input type="submit" name="_eventId_phoneEntered"
               value="Lookup Customer" />
    </form:form>
</body>
</html>
```

Эта простая форма предлагает клиенту ввести номер телефона. Но форма имеет два специальных ингредиента, позволяющих ей управлять выполнением последовательности.

Первый – скрытое поле `_flowExecutionKey`. При входе в состояние представление выполнение последовательности приостанавливается до того момента, пока пользователь не выполнит какого-либо действия. Представлению передается ключ выполнения последовательности, своеобразное «удостоверение» для последовательности. Когда пользователь отправит форму, ключ выполнения последовательности будет отправлен вместе с ней, в поле `_flowExecutionKey`, и последовательность продолжит выполнение с того места, где она была приостановлена.

Обратите также внимание на значение атрибута `name` кнопки `submit`. Часть `_eventId_` в значении атрибута – это подсказка фреймворку Spring Web Flow, что оставшаяся часть значения является именем события, которое должно быть возбуждено. Когда форма отправляется щелчком на этой кнопке, возбуждается событие `phoneEntered`, заставляющее выполнить переход к состоянию `lookupCustomer`.

Поиск информации о клиенте

После того как форма с приветствием будет отправлена на сервер, в ее составе будет отправлен и номер телефона клиента, которым можно воспользоваться, чтобы отыскать информацию о клиенте. Поиск выполняется элементом `<evaluate>` состояния `lookupCustomer`. Он извлекает номер телефона из параметров запроса и передает его методу `lookupCustomer()` компонента `pizzaFlowActions`.

В данный момент реализация метода `lookupCustomer()` не имеет большого значения. Достаточно знать, что он либо возвращает объект `Customer`, либо возбуждает исключение `CustomerNotFoundException`.

В первом случае объект `Customer` сохраняется в переменной `customer` (определяется атрибутом `result`) и выполняется переход по умолчанию к состоянию `customerReady`. Но если клиент не найден, метод возбудит исключение `CustomerNotFoundException`, и последовательность перейдет к состоянию `registrationForm`.

Регистрация нового клиента

В состоянии `registrationForm` пользователю предлагается указать адрес доставки. Подобно другим состояниям-представлениям, это состояние также отображает JSP-страницу, представленную в листинге 9.6.

Листинг 9.6. Регистрация нового клиента

```
<html xmlns:c="http://java.sun.com/jsp/jstl/core"
      xmlns:jsp="http://java.sun.com/JSP/Page"
      xmlns:spring="http://www.springframework.org/tags"
      xmlns:form="http://www.springframework.org/tags/form">

    <jsp:output omit-xml-declaration="yes"/>
    <jsp:directive.page contentType="text/html;charset=UTF-8" />

    <head><title>Spizza</title></head>

    <body>
        <h2>Customer Registration</h2>

        <form:form commandName="customer">
            <input type="hidden" name="_flowExecutionKey"
                  value="${flowExecutionKey}" />
            <b>Phone number:</b><form:input path="phoneNumber"/><br/>
            <b>Name:</b><form:input path="name"/><br/>
        </form:form>
    </body>
</html>
```

```

<b>Address: </b><form:input path="address"/><br/>
<b>City: </b><form:input path="city"/><br/>
<b>State: </b><form:input path="state"/><br/>
<b>Zip Code: </b><form:input path="zipCode"/><br/>
<input type="submit" name="_eventId_submit"
       value="Submit" />
<input type="submit" name="_eventId_cancel"
       value="Cancel" />
</form:form>
</body>
</html>

```

Это не первая форма, встретившаяся нам в нашей последовательности. Представление `welcome` тоже отображает форму, более простую, содержащую единственное поле, значение которого совсем несложно было извлечь из параметров запроса. Форма регистрации имеет более сложную организацию.

Чтобы избавиться от необходимости извлекать значения полей по отдельности из параметров запроса, имеет смысл связать форму с объектом `Customer` и переложить всю рутинную работу на фреймворк.

Проверка зоны обслуживания

Когда клиент введет адрес, необходимо убедиться, что он живет в пределах досягаемости. Если пиццерия Spizza окажется не в состоянии выполнить доставку, необходимо предупредить клиента и предложить ему прийти за заказом самому.

Принятие решения реализуется состоянием-решением. Состояние `checkDeliveryArea` имеет элемент `<if>`, который передает почтовый индекс клиента методу `checkDeliveryArea()` компонента `pizzaFlowActions`. Этот метод возвращает логическое значение: `true`, если клиент находится в пределах досягаемости, и `false` – в противном случае.

Если клиент находится в пределах досягаемости, последовательность выполняет переход к состоянию `addCustomer`. Иначе – к состоянию `deliveryWarning`. Представление, которое стоит за состоянием `deliveryWarning`, определено в файле `/WEB-INF/flows/pizza/customer/deliveryWarning.jspx` и приводится в листинге 9.7.

Листинг 9.7. Предупреждение о невозможности доставки пиццы по указанному адресу

```

<html xmlns:jsp="http://java.sun.com/JSP/Page">
<jsp:output omit-xml-declaration="yes"/>

```

```
<jsp:directive.page contentType="text/html;charset=UTF-8" />

<head><title>Spizza</title></head>

<body>
    <h2>Delivery Unavailable</h2>

    <p>The address is outside of our delivery area. You may
       still place the order, but you will need to pick it up
       yourself.</p>

    <![CDATA[
        <a href="${flowExecutionUrl}&_eventId=accept">
            Continue, I'll pick up the order</a> |
        <a href="${flowExecutionUrl}&_eventId=cancel">Never mind</a>
    ]]>
</body>
</html>
```

Ключевыми элементами страницы `deliveryWarning.jspx`, имеющими отношение к последовательности, являются две ссылки, позволяющие клиенту продолжить оформление заказа или отказаться от него. С помощью той же переменной `flowExecutionUrl`, что использовалась в состоянии `welcome`, эти ссылки возбуждают в последовательности событие `accept` или `cancel`. При получении события `accept` последовательность выполнит переход к состоянию `addCustomer`. Иначе будет выполнен глобальный переход `cancel`, и подпоследовательность попадет в состояние `cancel`.

О конечных состояниях будет говориться чуть ниже, а пока обсудим состояние `addCustomer`.

Сохранение информации о клиенте

К моменту, когда будет достигнуто состояние `addCustomer`, клиент уже введет свой адрес. Этот адрес необходимо сохранить на будущее (может быть, в базе данных). Состояние `addCustomer` имеет элемент `<evaluate>`, вызывающий метод `addCustomer()` компонента `pizzaFlow-Actions` и передающий ему переменную `customer` последовательности.

После вычисления выражения выполняется переход по умолчанию к конечному состоянию с идентификатором `customerReady`.

Завершение выполнения последовательности

Обычно конечные состояния не содержат ничего интересного. Но в данной последовательности не одно, а целых два конечных со-



стояния. Когда подпоследовательность завершается, она возбуждает событие в основной последовательности, идентификатор которого соответствует идентификатору конечного состояния. Если последовательность имеет единственное конечное состояние, она всегда будет возбуждать одно и то же событие. Но при наличии двух и более конечных состояний подпоследовательность получает возможность управлять дальнейшим выполнением вызывающей последовательности.

Когда последовательность идентификации клиента завершит выполнение обычным образом, она достигнет конечного состояния с идентификатором `customerReady`. Когда управление будет передано вызвавшей ее основной последовательности, она примет событие `customerReady`, в результате которого будет выполнен переход к состоянию `buildOrder`.

Обратите внимание, что конечное состояние `customerReady` включает элемент `<output>`. В последовательностях этот элемент играет роль инструкции `return`. Он возвращает вызывающей последовательности данные из подпоследовательности. В данном случае элемент `<output>` возвращает переменную `customer`, которую состояние `identifyCustomer` в главной последовательности сохраняет в заказе.

С другой стороны, если в какой-то момент в последовательности идентификации клиента будет возбуждено событие `cancel`, она совершил выход через конечное состояние с идентификатором `cancel`. Это приведет к возбуждению события `cancel` в главной последовательности и к переходу (через глобальный переход) к конечному состоянию.

9.3.3. Оформление заказа

На следующем этапе в главной последовательности, после идентификации клиента, выясняется сорт заказываемой пиццы. Подпоследовательность оформления заказа, диаграмма которой представлена на рис. 9.4, запрашивает у пользователя сорт пиццы и добавляет ее в заказ.

Как показано на рис. 9.4, центральное положение в подпоследовательности оформления заказа занимает состояние `showOrder`. Это первое состояние, которое видит пользователь после входа в последовательность, и в это же состояние он попадает после добавления очередной пиццы. Оно отображает текущее состояние заказа и предлагает добавить в заказ еще одну пиццу.

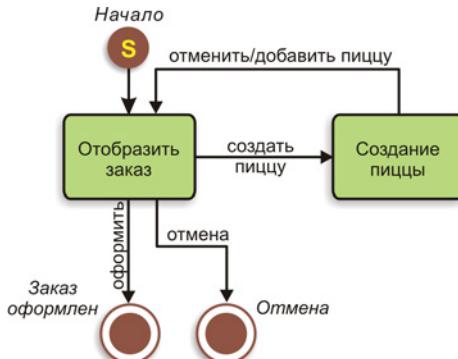


Рис. 9.4. Подпоследовательность добавления пиццы в заказ

После того как пользователь щелкнет на кнопке, чтобы добавить пиццу в заказ, последовательность перейдет к состоянию `createPizza`. Это еще одно состояние-представление, дающее возможность выбрать размер пиццы и добавки. С этого момента у пользователя появляется возможность выбирать – добавить еще одну пиццу или отменить заказ. В любом случае последовательность вернется в состояние `showOrder`.

Находясь в состоянии `showOrder`, пользователь может подтвердить заказ или отменить его. В обоих случаях подпоследовательность оформления заказа перейдет в конечное состояние, но от сделанного выбора зависит путь дальнейшего выполнения главной последовательности.

Как эта диаграмма выражается в терминах определения последовательности Spring Web Flow, показано в листинге 9.8.

Листинг 9.8. Состояния-представления в подпоследовательности оформления заказа

```

<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
      http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <input name="order" required="true" />

    <view-state id="showOrder">
  
```



```
<transition on="createPizza" to="createPizza" />
<transition on="checkout" to="orderCreated" />
<transition on="cancel" to="cancel" />
</view-state>

<view-state id="createPizza" model="flowScope.pizza">
    <on-entry>
        <set name="flowScope.pizza"
            value="new com.springinaction.pizza.domain.Pizza()" />

        <evaluate result="viewScope.toppingsList" expression=
            "T(com.springinaction.pizza.domain.Topping).asList()" />
    </on-entry>
    <transition on="addPizza" to="showOrder">
        <evaluate expression="order.addPizza(flowScope.pizza)" />
    </transition>
    <transition on="cancel" to="showOrder" />
</view-state>

<end-state id="cancel" />
<end-state id="orderCreated" />
</flow>
```

Эта подпоследовательность оперирует объектом Order, созданным в главной последовательности. То есть необходим некоторый способ передачи объекта Order из главной последовательности в подпоследовательность. В листинге 9.1 для этой цели использовался элемент `<input>`. Здесь этот же элемент используется для приема объекта Order. Если провести аналогии этой подпоследовательности с методами в языке Java, элемент `<input>` в данном случае фактически определяет сигнатуру подпоследовательности. Эта последовательность принимает один обязательный параметр с именем order.

Далее следует определение состояния showOrder, основное состояние-представление с тремя различными переходами: один ведет к созданию пиццы, один – к отправке заказа и один – к отмене заказа.

Состояние createPizza гораздо интереснее. Это представление является формой, добавляющей новый объект Pizza в заказ. Элемент `<on-entry>` добавляет в область видимости последовательности новый объект Pizza, который будет заполнен после отправки формы пользователем. Обратите внимание, что атрибут `model` этого состояния ссылается на тот же самый объект Pizza. Он будет связан с формой создания пиццы, представленной в листинге 9.9 ниже.

Листинг 9.9. Форма добавления пиццы в заказ, связанная с объектом в области видимости последовательности

```
<div xmlns:form="http://www.springframework.org/tags/form"
      xmlns:jsp="http://java.sun.com/JSP/Page">

    <jsp:output omit-xml-declaration="yes"/>
    <jsp:directive.page contentType="text/html;charset=UTF-8" />

    <h2>Create Pizza</h2>
    <form:form commandName="pizza">
        <input type="hidden" name="_flowExecutionKey"
               value="${flowExecutionKey}" />

        <b>Size: </b><br/>
        <form:radioButton path="size"
                          label="Small (12-inch)" value="SMALL"/><br/>
        <form:radioButton path="size"
                          label="Medium (14-inch)" value="MEDIUM"/><br/>
        <form:radioButton path="size"
                          label="Large (16-inch)" value="LARGE"/><br/>
        <form:radioButton path="size"
                          label="Ginormous (20-inch)" value="GINORMOUS"/>
        <br/>
        <br/>

        <b>Toppings: </b><br/>
        <form:checkboxes path="toppings" items="${toppingsList}"
                         delimiter="&lt;br&gt;"/><br/><br/>

        <input type="submit" class="button"
               name="_eventId_addPizza" value="Continue"/>
        <input type="submit" class="button"
               name="_eventId_cancel" value="Cancel"/>
    </form:form>
</div>
```

Когда форма будет отправлена щелчком на кнопке **Continue** (Продолжить), размер пиццы и выбранные добавки будут сохранены в объекте Pizza, и последовательность выполнит переход addPizza. Элемент `<evaluate>`, связанный с этим переходом, указывает, что объект Pizza должен быть передан в вызов метода `addPizza()` объекта order перед переходом к состоянию showOrder.



В этой последовательности имеются два пути, ведущих к завершению. В представлении `showOrder` пользователь может щелкнуть на кнопке **Cancel** (Отмена) или **Checkout** (Оформить). В любом случае последовательность перейдет в одно из двух конечных состояний. Атрибут `id` конечного состояния определяет событие, которое будет возбуждено, и в конечном счете следующий этап в главной последовательности. Главная последовательность выполнит переход либо к состоянию `cancel`, либо к состоянию `orderCreated`.

В первом случае внешняя последовательность завершит выполнение. Во втором – перейдет к состоянию-подпоследовательности `takePayment`, рассматриваемому ниже.

9.3.4. Прием оплаты

Никто не будет раздавать пиццы бесплатно, и пиццерия Spizza не смогла бы долго оставаться в этом бизнесе, если бы клиенты не оплачивали ее услуги. Наша главная последовательность приближается к концу, и ей осталось выполнить последнюю подпоследовательность, определяющую способ оплаты.

Эта простая последовательность изображена на рис. 9.5. Подобно подпоследовательности оформления заказа, данная подпоследовательность также принимает объект `Order` с помощью элемента `<input>`.

Как показано на рис. 9.5, после входа в подпоследовательность приема оплаты пользователь достигает состояния `takePayment`. В этом состоянии-представлении пользователь может указать, как он будет производить оплату: кредитной картой, чеком или наличными. Пос-

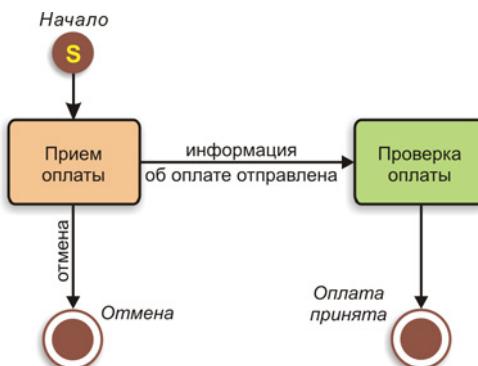


Рис. 9.5. Последний этап в процедуре заказа пиццы – принять оплату с помощью подпоследовательности

ле отправки информации об оплате последовательность переходит в состояние-действие `verifyPayment`, проверяющее допустимость выбранного способа оплаты.

В листинге 9.10 представлено определение подпоследовательности приема оплаты.

Листинг 9.10. Подпоследовательность приема оплаты содержит одно состояние-представление и одно состояние-действие

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
                           http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <input name="order" required="true"/>

    <view-state id="takePayment" model="flowScope.paymentDetails">
        <on-entry>
            <set name="flowScope.paymentDetails"
                 value="new com.springinaction.pizza.domain.PaymentDetails()"/>

            <evaluate result="viewScope.paymentTypeList" expression=
                "T(com.springinaction.pizza.domain.PaymentType).asList()"/>
        </on-entry>
        <transition on="paymentSubmitted" to="verifyPayment" />
        <transition on="cancel" to="cancel" />
    </view-state>

    <action-state id="verifyPayment">
        <evaluate result="order.payment" expression=
            "pizzaFlowActions.verifyPayment(flowScope.paymentDetails)"/>
        <transition to="paymentTaken" />
    </action-state>

    <end-state id="cancel" />
    <end-state id="paymentTaken" />
</flow>
```

При входе последовательности в состояние `takePayment` элемент `<on-entry>` настраивает форму приема оплаты, выполняя первое выражение на языке SpEL, создающее новый экземпляр `PaymentDetails` в области видимости последовательности. Этот объект будет хранить информацию, переданную с формой. Затем он создает в области ви-

димости представления переменную paymentTypeList со списком значений из перечисления PaymentType (определяется в листинге 9.11). Здесь оператор T() языка SpEL используется для получения класса PaymentType, используемого для вызова статического метода toList().

Листинг 9.11. Перечисление PaymentType, определяющее виды оплаты

```
package com.springinaction.pizza.domain;

import static org.apache.commons.lang.WordUtils.*;

import java.util.Arrays;
import java.util.List;

public enum PaymentType {
    CASH, CHECK, CREDIT_CARD;

    public static List<PaymentType> asList() {
        PaymentType[] all = PaymentType.values();
        return Arrays.asList(all);
    }

    @Override
    public String toString() {
        return capitalizeFully(name().replace('_', ' '));
    }
}
```

После получения формы оплаты заказа пользователь может либо отправить информацию об оплате, либо отменить заказ. В зависимости от выбора клиента подпоследовательность приема оплаты завершается либо в конечном состоянии paymentTaken, либо в конечном состоянии cancel. Подобно другим подпоследовательностям, по достижении конечного состояния данная подпоследовательность возвращает управление главной последовательности, а значение атрибута id конечного состояния определяет, куда будет выполнен переход в главной последовательности.

На данный момент мы исследовали все пути выполнения главной последовательности и ее подпоследовательностей. Мы познакомились со многими особенностями фреймворка Spring Web Flow. Но, прежде чем оставить тему веб-последовательностей, рассмотрим способы обеспечения безопасности доступа к последовательностям и к их состояниям.

9.4. Безопасность веб-последовательностей

В следующей главе будут рассматриваться проблемы обеспечения безопасности приложений на основе фреймворка Spring с использованием Spring Security. Но, пока мы не закончили обсуждение темы применения фреймворка Spring Web Flow, рассмотрим поддержку обеспечения безопасности на уровне последовательностей, которую осуществляет Spring Web Flow, помимо Spring Security.

Состояния, переходы и даже целые последовательности можно обезопасить средствами фреймворка Spring Web Flow, добавляя элемент `<secured>` в другие элементы. Например, чтобы обезопасить доступ к состоянию-представлению, в определение состояния можно добавить элемент `<secured>`, как показано ниже:

```
<view-state id="restricted">
    <secured attributes="ROLE_ADMIN" match="all"/>
</view-state>
```

Согласно этому определению, доступ к состоянию-представлению будет предоставлен только пользователям с привилегиями `ROLE_ADMIN` (в соответствии со значением атрибута `attributes`). В атрибуте `attributes` можно через запятую указать список привилегий, которыми должны обладать пользователи, чтобы получить доступ к состоянию, переходу или последовательности. В атрибуте `match` можно указать значение `any` или `all`. Если атрибут `match` имеет значение `any`, пользователь должен обладать хотя бы одной привилегией из списка в атрибуте `attributes`. Если атрибут `match` имеет значение `all`, пользователь должен обладать всеми указанными привилегиями.

У кого-то может возникнуть вопрос: «Как пользователь получит привилегии, проверяемые элементом `<secured>`? Имеет ли это отношение к процедуре регистрации пользователя в приложении?». Ответы на эти вопросы вы найдете в следующей главе.

9.5. Интеграция Spring Web Flow с другими фреймворками

В этой главе мы занимались созданием веб-приложения на основе фреймворка Spring Web Flow, входящего в состав Spring MVC. Однако, возможно, кому-то будет интересно узнать, что Spring Web

Flow не требует обязательного использования Spring MVC. Фактически фреймворк Spring Web Flow обладает встроенной поддержкой следующих фреймворков:

- Jakarta Struts;
- JavaServer Faces;
- Spring Portlet MVC.

В списке проблем фреймворка Spring Web Flow открытой остается проблема интеграции с фреймворком WebWork 2 (который, по всей видимости, также должен работать в Struts 2). Познакомиться с проблемами интеграции с фреймворком WebWork, а также с состоянием дел в этом направлении можно по адресу: <http://opensource.atlassian.com/projects/spring/browse/SWF-76>.

Независимо от того, какой веб-фреймворк будет выбран для встраивания последовательностей, определение этих последовательностей останется переносимым для всех поддерживаемых фреймворков. Единственное отличие между ними составляют точки интеграции (например, в приложениях на основе Spring MVC используется компонент `FlowController`, а в приложениях на основе Struts – компонент `FlowAction`).

Прежде чем завершить обсуждение фреймворка Spring Web Flow, посмотрим, как Spring Web Flow обеспечивает поддержку интеграции с фреймворком JSF.

9.5.1. JavaServer Faces

Чтобы иметь возможность определять последовательности Spring Web Flow в приложениях на основе фреймворка JSF, в файле `faces-config.xml` необходимо настроить некоторые элементы Spring Web Flow. Эти настройки демонстрируются в следующем фрагменте файла `faces-config.xml`:

```
<application>
    <navigation-handler>
        org.springframework.webflow.executor.jsf.FlowNavigationHandler
    </navigation-handler>
    <property-resolver>
        org.springframework.webflow.executor.jsf.FlowPropertyResolver
    </property-resolver>
    <variable-resolver>
        org.springframework.webflow.executor.jsf.FlowVariableResolver
    </variable-resolver>
```

```
</variable-resolver>
<variable-resolver>
    org.springframework.web.jsf.DelegatingVariableResolver
</variable-resolver>
<variable-resolver>
    org.springframework.web.jsf.WebApplicationContextVariableResolver
</variable-resolver>
</application>
```

Навигация в JSF обычно реализуется с применением элементов `<navigation-rule>` в файле `faces-config.xml`. Однако в Spring Web Flow навигация реализуется с применением элементов `<transition>`. Поэтому, чтобы обеспечить интеграцию фреймворка Spring Web Flow с JSF, необходимо настроить компонент `FlowNavigationHandler`, обработчик механизма навигации, использующий определение последовательности для реализации навигации в пределах последовательности (с возвратом к использованию компонента `NavigationHandler` при выполнении вне последовательности).

Кроме того, нам может понадобиться обращаться из JSF к переменным и их свойствам в области видимости последовательности. К сожалению, механизм JSF доступа к переменным и свойствам по умолчанию ничего не знает о переменных в области видимости последовательностей. Однако в Spring Web Flow имеются компоненты `FlowVariableResolver` и `FlowPropertyResolver`, позволяющие фреймворку JSF отыскивать данные последовательностей, снабженные префиксом `flowScope` (например, `flowScope.order.total`).

Кроме того, необходимо настроить еще два механизма разрешения переменных: компоненты `DelegatingVariableResolver` и `WebApplicationContextVariableResolver`. Эти компоненты удобно использовать для обращения к переменным JSF как к компонентам из контекста приложения Spring.

Наконец, для интеграции фреймворка Spring Web Flow с JSF необходимо добавить еще одну настройку в файл `faces-config.xml`. Информация, касающаяся выполнения последовательности, должна сохраняться между запросами и восстанавливаться в начале обработки каждого запроса. Для этого можно использовать компонент `FlowPhaseListener`, входящий в состав Spring Web Flow, который является реализацией интерфейса `PhaseListener` в JSF и управляет выполнением последовательности от имени JSF. Ниже приводится пример настройки компонента в файле `faces-config.xml`:

```
<lifecycle>
    <phase-listener>
        org.springframework.webflow.executor.jsf.FlowPhaseListener
    </phase-listener>
</lifecycle>
```

Компонент `FlowPhaseListener` решает три задачи, каждая из которых связана с определенным этапом в жизненном цикле JSF:

- ❑ на этапе `BEFORE_RESTORE_VIEW` компонент `FlowPhaseListener` восстанавливает информацию о выполнении последовательности, опираясь на идентификатор в аргументах запроса (если указан), определяющий блок информации для выполняемой последовательности;
- ❑ на этапе `BEFORE_RENDER_RESPONSE` компонент `FlowPhaseListener` создает новый идентификатор сохраняемого блока информации для выполняемой последовательности;
- ❑ на этапе `AFTER_RENDER_RESPONSE` компонент `FlowPhaseListener` сохраняет блок информации для выполняемой последовательности, используя идентификатор, генерированный на этапе `BEFORE_RENDER_RESPONSE`.

После добавления в файл `faces-config.xml` всех необходимых определений можно начинать пользоваться последовательностями. Следующий фрагмент JSF создает ссылку, запускающую последовательность с именем `Order-flow`:

```
<h:commandLink
    value="Order Pizza"
    action="flowId:Order-flow"/>
```

9.6. В заключение

Не все веб-приложения допускают возможность свободного перемещения по своим страницам. Иногда необходимо вести пользователя за руку, задавать вопросы и в зависимости от ответов возвращать соответствующие страницы. Работа с подобными приложениями больше напоминает диалог между приложением и пользователем, чем выбор из множества вариантов в меню.

В этой главе мы исследовали веб-фреймворк `Spring Web Flow`, дающий возможность создавать диалоговые приложения. Попутно мы разработали приложение оформления заказа пиццы на осно-

ве последовательностей. Сначала мы определили общий процесс оформления заказа, начинающийся со сбора информации о клиенте и завершающийся сохранением заказа в системе.

Последовательность конструируется из различных состояний и переходов, определяющих, как будет протекать диалог от состояния к состоянию. Сами состояния делятся на несколько категорий: состояния-действия, выполняющие некоторую логику; состояния-представления, вовлекающие пользователя в диалог; состояния-решения, динамически управляющие выполнением последовательности; и конечные состояния, обозначающие конец последовательности. Кроме того, существуют состояния-подпоследовательности, которые сами определяют последовательности.

Наконец, было показано, как можно обезопасить последовательности, состояния и переходы, разрешив доступ к ним только пользователям, обладающим определенными привилегиями. Но обсуждение вопросов аутентификации пользователей в приложении и приобретение ими необходимых привилегий было отложено на потом. В этом нам может помочь фреймворк Spring Security, исследованием которого мы займемся в следующей главе.



Глава 10. Безопасность в Spring

В этой главе рассматриваются следующие темы:

- ❑ введение в Spring Security;
- ❑ обеспечение безопасности с помощью сервлет-фильтров;
- ❑ аутентификация с использованием баз данных и LDAP;
- ❑ обеспечение безопасности вызовов методов.

Доводилось ли вам замечать, что большинство персонажей в телевизионных комедийных сериалах никогда не запирают свои двери на замок? Причем постоянно. Например, в сериале «Seinfeld» («Сайнфельд») Крамер (Kramer) часто позволяет себе заходить в квартиру к Джерри (Jerry) и одалживать у него вещи и продукты. В сериале «Friends» («Друзья») разные персонажи часто входят друг к другу в комнату без предупреждения, совершенно не задумываясь. Однажды, находясь в Лондоне, Росс (Ross) ворвался в комнату в отеле, где поселился Чендлер (Chandler), и застал его в пикантной ситуации со своей сестрой.

В сериале «Leave it to Beaver» («Проделки Бивера») персонажи тоже нечасто запирали свои двери. Довольно странно видеть, как герои сериалов позволяют бесцеремонно вторгаться в свои дома и квартиры, в то время когда все вокруг заботятся о неприкосновенности частной жизни.

Очень грустно, что в мире существуют злоумышленники, стремящиеся завладеть нашими деньгами, богатствами, автомобилями и другими ценностями. И совершенно неудивительно, что находятся мошенники, ищащие возможность украсть наши данные, взламывая незащищенные приложения, поскольку информация является, пожалуй, самой большой ценностью.

Как разработчики программного обеспечения мы должны принимать меры по защите информации, имеющейся в наших приложениях. Будь то электронный почтовый ящик, защищенный именем пользователя и паролем, или брокерский счет, защищенный персональным идентификационным кодом, безопасность является критически важным *аспектом* большинства приложений.

Выше я не случайно использовал слово «аспект» применительно к безопасности приложений. Обеспечение безопасности – это задача,

выходящая за рамки функциональности приложения. По большей части, приложение не должно участвовать в обеспечении безопасности самого себя. Разумеется, вы можете встраивать реализацию приемов обеспечения безопасности непосредственно в прикладной программный код (и в этом нет ничего необычного), но гораздо лучше отделять задачи обеспечения безопасности от прикладных задач.

Если из моих слов вам показалось, что безопасность можно обеспечить с использованием приемов аспектно-ориентированного программирования, то вы совершенно правы. В этой главе мы исследуем способы обеспечения безопасности приложений с помощью аспектов. Но мы не будем заниматься разработкой этих аспектов, вместо этого мы задействуем Spring Security – фреймворк обеспечения безопасности, реализованный с применением механизмов Spring AOP и сервлет-фильтров¹.

10.1. Введение в Spring Security

Spring Security – это фреймворк обеспечения безопасности, предоставляющий возможность декларативного управления безопасностью приложений на основе фреймворка Spring. Фреймворк Spring Security представляет собой всеобъемлющее решение по обеспечению безопасности, реализующее возможность аутентификации и авторизации как на уровне веб-запросов, так и на уровне вызовов методов. Опираясь на возможности фреймворка Spring Framework, Spring Security в полной мере использует поддержку *внедрения зависимостей* (DI) и *аспектно-ориентированного программирования*.

Проект Spring Security начал разрабатываться под названием *Acegi Security*. Acegi – это мощный фреймворк обеспечения безопасности, но он имел один существенный недостаток – большой объем конфигурационного файла в формате XML. Не буду загружать вас демонстрацией примера такого конфигурационного файла, скажу лишь, что типичная конфигурация Acegi занимала несколько сотен строк разметки XML.

В версии 2.0 фреймворк Acegi Security был переименован в Spring Security. Но версия 2.0 отличается не только названием. В Spring

¹ Возможно, я получу массу электронных писем с возражениями, но я все-таки скажу, что сервлет-фильтры являются примитивной формой AOP, где шаблоны URL выступают в роли языка выражений для определения множеств точек внедрения. Уф-ф... Я сказал это... Теперь мне стало легче.



Security 2.0 появилось новое пространство имен XML, связанное с безопасностью и предназначенное для настройки системы безопасности в Spring. Новое пространство имен, наряду с аннотациями и обоснованными настройками по умолчанию, позволило сократить объем конфигурации безопасности с сотен до десятков и даже единиц строк. В самой свежей версии, Spring Security 3.0, была добавлена поддержка языка выражений SpEL, что еще больше упростило настройку безопасности.

Фреймворк Spring Security обеспечивает безопасность с двух сторон. Для ограничения доступа и обеспечения безопасности на уровне запросов в Spring Security используются сервлет-фильтры. А для обеспечения безопасности на уровне вызовов методов с использованием Spring AOP фреймворк Spring Security предоставляет объекты-обертки и позволяет применять советы, гарантирующие авторизацию пользователей.

10.1.1. Обзор Spring Security

Независимо от типа приложения, которое предполагается обезопасить с применением Spring Security, в первую очередь следует добавить модули Spring Security в библиотеку классов (classpath) приложения. Версия Spring Security 3.0 делится на восемь *модулей*, которые перечислены в табл. 10.1.

Таблица 10.1. Spring Security делится на восемь модулей

Модуль	Описание
ACL	Обеспечивает поддержку безопасности доменных объектов с использованием списков управления доступом (Access Control Lists, ACL)
CAS Client	Обеспечивает интеграцию с централизованной службой аутентификации JA-SIG (Central Authentication Service, CAS)
Configuration	Обеспечивает поддержку пространства имен XML
Core	Основная библиотека Spring Security
LDAP	Обеспечивает поддержку аутентификации с использованием облегченного протокола доступа к каталогу (Lightweight Directory Access Protocol, LDAP)
OpenID	Обеспечивает интеграцию с децентрализованной службой OpenID
Tag Library	Включает множество тегов JSP для обеспечения безопасности на уровне представлений
Web	Обеспечивает поддержку безопасности веб-приложений с применением фильтров

В библиотеку классов приложения требуется включить, как минимум, модули Core и Configuration. Фреймворк Spring Security часто используется для обеспечения безопасности веб-приложений. Это напрямую относится к приложению Spitter, поэтому нам также следует подключить модуль Web. Было бы желательно иметь возможность воспользоваться преимуществами поддержки JSP-тегов в Spring Security, поэтому подключим также и этот модуль.

Теперь можно приступать к декларативной настройке безопасности в Spring Security. Для начала познакомимся с пространством имен XML, предоставляемым фреймворком Spring Security.

10.1.2. Использование конфигурационного пространства имен Spring Security

Когда фреймворк Spring Security еще носил название Acegi Security, все элементы поддержки безопасности настраивались как компоненты в контексте приложения Spring. Типичный конфигурационный файл Acegi мог содержать десятки объявлений компонентов <bean> и располагаться на множестве страниц.

Фреймворк Spring Security поддерживает специализированное пространство имен, существенно упрощающее настройку безопасности в Spring. Это новое пространство имен, наряду с обоснованными настройками по умолчанию, уменьшает размер типичного конфигурационного XML-файла с сотен до десятков строк.

Единственное, что необходимо сделать, чтобы получить возможность пользоваться новым пространством имен, – подключить его в XML-файле, добавив его объявление, как показано в листинге 10.1:

Листинг 10.1. Добавление пространства имен Spring Security в конфигурационный XML-файл Spring

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    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-3.0.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security-3.0.xsd">

    <!-- Здесь располагаются элементы с префиксом security: --&gt;

&lt;/beans&gt;</pre>

---


```

В приложении Spitter мы поместим все настройки безопасности в отдельный конфигурационный файл с именем `spitter-security.xml`. Поскольку в этом файле все элементы будут принадлежать специализированному пространству имен, его можно объявить пространством имен по умолчанию, как показано в листинге 10.2.

Листинг 10.2. Объявление пространства имен настройки безопасности пространством имен по умолчанию

```
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns="http://www.springframework.org/schema/security"
    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-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.0.xsd">

    <!-- Здесь располагаются элементы без префикса security: -->

</beans:beans>
```

Объявив пространство имен элементов настройки безопасности пространством имен по умолчанию, можно избежать необходимости использовать префикс `security` во всех элементах.

Теперь, когда появилась возможность аккуратно расположить все настройки безопасности в отдельном месте, можно приступать к добавлению в приложение Spitter настроек безопасности на уровне веб-запросов.

10.2. Безопасность веб-запросов

Все взаимодействия с веб-приложениями на языке Java начинаются в компоненте `HttpServletRequest`. И коль скоро средством доступа к веб-приложению является запрос, то с него и следует начинать обеспечивать безопасность.

Обычно настройка безопасности на уровне запросов начинается с объявления одного или более шаблонов URL-адресов, требующих некоторых привилегий, и ограничения доступа к содержимому по этим адресам для пользователей, не обладающих необходимыми привилегиями. Более того, доступ к некоторым URL можно также ограничить протоколом HTTPS.

Прежде чем ограничивать доступ пользователям, не обладающим необходимыми привилегиями, необходимо предусмотреть возмож-

ность определять, кто пользуется приложением. Поэтому приложение должно аутентифицировать пользователя, предложив ему идентифицировать себя.

Фреймворк Spring Security поддерживает эти и многие другие способы обеспечения безопасности на уровне запросов. Но для начала следует настроить сервлет-фильтры, предоставляющие разнообразные возможности.

10.2.1. Сервлет-фильтры

Для обеспечения различных аспектов безопасности Spring Security использует несколько *сервлет-фильтров*. Кому-то может показаться, что это означает необходимость добавлять в конфигурационный файл `web.xml` приложения несколько элементов `<filter>`. Но это не так – благодаря волшебству Spring достаточно настроить только один фильтр. В частности, достаточно добавить следующий элемент `<filter>`:

```
<filter>
    <filter-name>springSecurityFilterChain</filter-name>
    <filter-class>
        org.springframework.web.filter.DelegatingFilterProxy
    </filter-class>
</filter>
```

`DelegatingFilterProxy` – это специальный сервлет-фильтр, не имеющий самостоятельного значения, так как он просто делегирует фильтрацию реализации интерфейса `javax.servlet.Filter`, зарегистрированной в контексте приложения Spring в виде компонента, как показано на рис. 10.1.



Рис. 10.1. Объекты-прокси `DelegatingFilterProxy`, обеспечивающие фильтрацию, делегируют фильтрацию компонентам в контексте приложения Spring

Чтобы задействовать фильтры из фреймворка Spring Security, в них необходимо внедрить какие-то другие компоненты. Однако фильтры, зарегистрированные в файле `web.xml`, нельзя внедрить

компоненты. Но с помощью DelegatingFilterProxy можно настроить фактические фильтры в Spring, воспользовавшись преимуществом поддержки внедрения зависимостей.

Значение в элементе `<filter-name>` играет важную роль. Это имя будет использоваться для поиска компонента фильтра в контексте приложения Spring. Фреймворк Spring Security автоматически создает компонент фильтра с идентификатором `springSecurityFilterChain`, потому что это имя указано в файле `web.xml`.

Что касается самого компонента `springSecurityFilterChain`, это еще один специальный фильтр, известный как `FilterChainProxy`. Это единый фильтр, объединяющий в себе один или более других фильтров. Для обеспечения различных возможностей фреймворк Spring Security опирается на разные сервлет-фильтры. Но вам практически никогда не придется задумываться об этой особенности, потому что вам едва ли когда-нибудь потребуется явно объявлять компонент `springSecurityFilterChain` или любые другие фильтры, объединяемые в нем. Фреймворк Spring Security автоматически создает все необходимые компоненты при настройке элемента `<http>`, о чем рассказывается ниже.

10.2.2. Минимальная настройка безопасности

В ранних версиях Spring Security для настройки основных механизмов системы безопасности требовалось добавлять в конфигурационные XML-файлы огромное количество элементов. Но в последних версиях Spring Security достаточно добавить следующий фрагмент:

```
<http auto-config="true">
    <intercept-url pattern="/**" access="ROLE_SPITTER" />
</http>
```

Эти три строчки обеспечивают перехват запросов для всех URL (как следует из значения атрибута `pattern` элемента `<intercept-url>`) и разрешают доступ к содержимому только аутентифицированным пользователям, обладающим привилегией `ROLE_SPITTER`. Элемент `<http>` автоматически настраивает компонент `FilterChainProxy` (которому делегирует фильтрацию объект `DelegatingFilterProxy`, настроенный в файле `web.xml`) и все остальные компоненты в цепочке, реализующие фильтрацию.

В дополнение к этим компонентам фильтров можно получить дополнительные возможности, установив атрибут `auto-config` в значение `true`. Автоматическая настройка включает в приложение страницу аутентификации, поддержку аутентификации средствами протокола HTTP и поддержку завершения сеанса работы. В действительности установка атрибута `auto-config` в значение `true` эквивалентна явной настройке этих возможностей, как показано ниже:

```
<http>
    <form-login />
    <http-basic />
    <logout />
    <intercept-url pattern="/**" access="ROLE_SPITTER" />
</http>
```

Познакомимся с ними поближе, чтобы понять, как ими пользоваться.

Аутентификация с помощью формы

Одним из преимуществ установки атрибута `auto-config` в значение `true` является автоматическое создание страницы аутентификации фреймворком Spring Security. В листинге 10.3 представлена разметка HTML этой формы.

Листинг 10.3. Фреймворк Spring Security может автоматически создавать простую форму аутентификации

```
<html>
    <head><title>Login Page</title></head>
    <body onload='document.f.j_username.focus();'>
        <h3>Login with Username and Password</h3>
        <form name='f' method='POST' action='/Spitter/j_spring_security_check'>    <!-- Путь к фильтру -->
            <table>                                <!-- аутентификации -->
                <tr><td>User:</td><td>      <!-- Поле ввода имени пользователя -->
                    <input type='text' name='j_username' value=''%gt;
                </td></tr>
                <tr><td>Password:</td><td> <!-- Поле ввода пароля -->
                    <input type='password' name='j_password' />
                </td></tr>
                <tr><td colspan='2'>
                    <input name="submit" type="submit"/>
                </td></tr>
```

```
<tr><td colspan='2'>
    <input name="reset" type="reset"/>
</td></tr>
</table>
</form>
</body>
</html>
```

Адрес URL автоматически сгенерированной формы аутентификации складывается из базового адреса URL приложения и относительного пути `/spring_security_login`. Например, при обращении к приложению Spitter на локальном компьютере этот URL будет иметь вид: http://localhost:8080/Spitter/spring_security_login.

На первый взгляд возможность автоматического создания формы аутентификации выглядит большим преимуществом. Но эта форма слишком проста и не отличается эстетической привлекательностью, поэтому в большинстве случаев она заменяется специально созданной формой.

Чтобы добавить в приложение свою форму аутентификации, необходимо настроить элемент `<form-login>`:

```
<http auto-config="true" use-expressions="false">
    <form-login login-processing-url="/static/j_spring_security_check"
        login-page="/login"
        authentication-failure-url="/login?login_error=t"/>
</http>
```

Атрибут `login` определяет новый относительный путь к странице аутентификации. В данном случае указывается, что страница аутентификации будет иметь относительный путь `/login`, который в конечном итоге обслуживается контроллером Spring MVC. Аналогично, в случае неудачной попытки аутентификации, атрибут `authentication-failure-url` будет отправлять пользователя обратно на страницу аутентификации.

Обратите внимание на значение `/static/j_spring_security_check` в атрибуте `login-processing-url`. Это URL, куда будет отправляться форма для аутентификации пользователя.

Даже если вы не планируете использовать автоматически сгенерированную форму аутентификации, знание особенностей ее функционирования будет совсем не лишним. Прежде всего мы знаем, что Spring Security обрабатывает запросы на аутентификацию по адрес-

су /\$spitter/j_spring_security_check. А также что имя пользователя и пароль должны отправляться в составе запроса, в виде параметров с именами j_username и j_password. Обладая этими знаниями, не составит труда создать собственную страницу аутентификации.

Новая страница аутентификации для приложения Spitter реализована в виде страницы JSP, которая обслуживается контроллером Spring MVC. Сама страница представлена ниже, в листинге 10.4.

Листинг 10.4. Приложение Spitter использует собственную JSP-страницу аутентификации

```
<%@ taglib prefix="s" uri="http://www.springframework.org/tags"%>
<div>
    <h2>Sign in to Spitter</h2>

    <p>
        If you've been using Spitter from your phone,
        then that's amazing...we don't support IM yet.
    </p>

    <!-- Путь к фильтру аутентификации -->
    <spring:url var="authUrl" value="/static/j_spring_security_check" />
    <form method="post" class="signin" action="${authUrl}">

        <fieldset>
            <table cellspacing="0">
                <tr>
                    <th><label for="username_or_email">Username or Email</label></th>
                    <td><input id="username_or_email"
                            name="j_username"
                            type="text" />      <!-- Поле ввода имени пользователя -->
                </td>
            </tr>
            <tr>
                <th><label for="password">Password</label></th>
                <td><input id="password"
                            name="j_password"
                            type="password" />  <!-- Поле ввода пароля -->
                <small><a href="/account/resend_password">Forgot?</a></small>
            </td>
        </tr>
        <tr>
            <th></th>
            <td><input id="remember_me"
```

```
        name="_spring_security_remember_me"
        type="checkbox"/>    <!-- Флажок "запомнить меня" -->
    <label for="remember_me"
           class="inline">Remember me</label></td>
</tr>
<tr>
    <th></th>
    <td><input name="commit" type="submit" value="Sign In" /></td>
</tr>
</table>
</fieldset>
</form>
<script type="text/javascript">
    document.getElementById('username_or_email').focus();
</script>
</div>
```

Эта страница аутентификации существенно отличается от страницы, генерируемой фреймворком Spring Security автоматически, тем не менее она также отправляет имя пользователя и пароль в параметрах `j_username` и `j_password`. Все остальное – лишь декорации.

Обратите также внимание на присутствие в листинге 10.4 флажка **remember me** (запомнить меня). Подробнее о том, как он действует, будет рассказываться в разделе 9.4.4. А сейчас познакомимся с поддержкой аутентификации посредством протокола HTTP.

Аутентификация средствами протокола HTTP

Аутентификация с использованием формы идеально подходит для людей, пользующихся приложением. Но в главе 11 будет показано, как превращать страницы веб-приложений в RESTful API, когда пользователем приложения является другое приложение и использовать форму аутентификации становится неудобно.

Механизм аутентификации, предусмотренный протоколом HTTP, обеспечивает возможность передачи информации об аутентификации непосредственно в HTTP-запросе. Возможно, вам приходилось сталкиваться с этой возможностью прежде. В браузере эта процедура выполняется с использованием простого модального диалога.

Но это лишь внешнее проявление действия механизма в веб-браузере. В действительности запрос на аутентификацию возвращается в виде ответа HTTP 401, указывающего на необходимость передать имя пользователя и пароль в теле запроса. Эта возмож-

ность отлично подходит для REST-клиентов, позволяя им аутентифицировать себя при пользовании услугами приложения.

Аутентификация средствами протокола HTTP включается с помощью элемента `<http-basic>` и не требует большого количества настроек – она может быть либо включена, либо выключена. Поэтому не будем больше останавливаться на этой теме и двинемся дальше, где нас ждет элемент `<logout>`.

Завершение сеанса работы

Элемент `<logout>` настраивает фильтр Spring Security, который будет закрывать сеанс работы с пользователем. При использовании с настройками по умолчанию элемент `<logout>` отображается на адрес `/j_spring_security_logout`. Но, чтобы устранить конфликт с настройками DispatcherServlet, необходимо переопределить URL фильтра, как это делалось для формы аутентификации. Для этого следует определить атрибут `logout-url`:

```
<logout logout-url="/static/j_spring_security_logout"/>
```

На этом завершается обсуждение возможностей, предоставляемых автоматически. Но исследование Spring Security не заканчивается. Познакомимся поближе с элементом `<intercept-url>` и посмотрим, как с его помощью управлять доступом на уровне запросов.

10.2.3. Перехват запросов

В предыдущем разделе был представлен простой пример использования элемента `<intercept-url>`. Но мы узнали о нем совсем немного... пока.

Элемент `<intercept-url>` – это первая линия обороны в системе безопасности. Его атрибут `pattern` определяет шаблон URL, который будет сопоставляться с входящими запросами. Если какой-либо запрос совпадет с шаблоном, к нему будут применены правила безопасности, определяемые элементом `<intercept-url>`.

Вернемся к определению элемента `<intercept-url>`, представленному выше:

```
<intercept-url pattern="/**" access="ROLE_SPITTER" />
```

По умолчанию в атрибуте `pattern` указывается шаблон пути в стиле утилиты Ant. Но если атрибуту `path-type` элемента `<http>` присво-



ить значение `regex`, в качестве шаблона пути можно будет использовать регулярное выражение.

В данном случае в атрибуте `pattern` задано значение `/**`, указывающее, что перехватываться должны все запросы, независимо от URL, и для получения доступа к содержимому пользователь должен обладать привилегией `ROLE_SPITTER`. Шаблон `/**` имеет широкую область значений и иногда бывает необходимо ограничить ее.

Представьте, что в приложении Spitter имеются специальные страницы, которые должны быть доступны только администраторам. Чтобы ограничить доступ к ним, можно добавить следующий элемент `<intercept-url>` перед уже имеющимся:

```
<intercept-url pattern="/admin/**" access="ROLE_ADMIN" />
```

Наш первый элемент `<intercept-url>` гарантирует, что к большей части приложения смогут обращаться только пользователи, обладающие привилегией `ROLE_SPITTER`, а данный элемент `<intercept-url>` ограничивает доступ к ветви `/admin` в иерархии сайта, допуская к ней только пользователей с привилегией `ROLE_ADMIN`.

В файле конфигурации допускается использовать любое количество элементов `<intercept-url>`, чтобы обезопасить различные пути в веб-приложении. Но важно помнить, что правила, определяемые элементами `<intercept-url>`, применяются в направлении сверху вниз. То есть этот новый элемент `<intercept-url>` должен находиться перед оригинальным элементом, иначе шаблон пути `/**` получит более высокий приоритет.

Настройка безопасности с применением выражений Spring

Настройка требуемых привилегий реализуется достаточно просто, но выглядит несколько однобоко. А что, если потребуется выразить ограничения, опираясь не только на имеющиеся привилегии?

В главе 2 было показано, как использовать язык выражений Spring (Spring Expression Language, SpEL) для связывания свойств компонентов. Начиная с версии 3.0, фреймворк Spring Security предоставляет возможность использовать SpEL для определения правил доступа. Чтобы воспользоваться ею, необходимо определить значение `true` в атрибуте `use-expressions` элемента `<http>`:

```
<http auto-config="true" use-expressions="true">
...
</http>
```

Теперь можно использовать выражения на языке SpEL в атрибуте access. Ниже показано, как на языке SpEL потребовать наличия привилегии ROLE_ADMIN при обращении к адресам, совпадающим с шаблоном /admin/**:

```
<intercept-url pattern="/admin/**" access="hasRole('ROLE_ADMIN')"/>
```

Этот элемент `<intercept-url>` полностью эквивалентен предыдущему, за исключением того, что в нем используется выражение на языке SpEL. Выражение `hasRole()` возвращает `true`, если текущий пользователь обладает указанной привилегией. Но `hasRole()` – лишь одно из множества поддерживаемых выражений, имеющих отношение к безопасности. В табл. 10.2 перечислены все выражения SpEL, поддерживаемые фреймворком Spring Security 3.0.

Таблица 10.2. Фреймворк Spring Security расширяет язык выражений Spring несколькими дополнительными выражениями

Пространство имен	Назначение
authentication	Объект аутентификации пользователя
denyAll	Всегда возвращает <code>false</code>
hasAnyRole(list_of_roles)	<code>true</code> , если пользователь обладает какой-либо из привилегий, перечисленных в списке <code>list_of_roles</code>
hasRole(role)	<code>true</code> , если пользователь обладает привилегией <code>role</code>
hasIpAddress(IP Address)	IP-адрес пользователя (доступен только в веб-приложениях)
isAnonymous()	<code>true</code> , если текущий пользователь не был аутентифицирован
isAuthenticated()	<code>true</code> , если текущий пользователь был аутентифицирован
isFullyAuthenticated()	<code>true</code> , если текущий пользователь был аутентифицирован и не использовал функцию «запомнить меня»
isRememberMe()	<code>true</code> , если текущий пользователь был аутентифицирован автоматически
permitAll	Всегда возвращает <code>false</code>
principal	Основной объект, представляющий пользователя

Благодаря поддержке языка выражений SpEL доступ к страницам можно ограничивать, основываясь не только на привилегиях пользователя. Например, если доступ к страницам администрирования должен ограничиваться не только наличием привилегии ROLE_ADMIN,



но и определенным IP-адресом, элемент <intercept-url> можно определить, как показано ниже:

```
<intercept-url pattern="/admin/**"
    access="hasRole('ROLE_ADMIN') and hasIpAddress('192.168.1.2')"/>
```

Поддержка языка SpEL обеспечивает практически неограниченные возможности настройки безопасности. Готов поспорить, что вы уже придумали интересные способы ограничения доступа с помощью SpEL.

А сейчас познакомимся с еще одной интересной особенностью элемента <intercept-url>: принудительное требование к используемому протоколу.

Принудительное использование протокола HTTPS

Передача данных по протоколу HTTP весьма небезопасна. Возможно, нет ничего страшного, если текст сообщений в приложении Spitter будет отправляться с помощью открытого протокола HTTP. Но передача секретной информации, такой как пароли и номера кредитных карт, по протоколу HTTP может привести к большим неприятностям. Именно поэтому секретную информацию лучше отправлять в зашифрованном виде, по протоколу HTTPS.

Задействовать протокол HTTPS довольно просто. Достаточно лишь добавить символ «s» после «http» в URL и все. Правильно?

Да, это так, но при этом вся ответственность за использование протокола HTTPS возлагается на программиста. А теперь представьте, что в приложении имеются десятки и сотни ссылок и форм, в которых должен быть указан протокол HTTPS. Слишком легко забыть добавить такую важную букву «s». Слишком велика вероятность, что программист пропустит ее в одном-двух местах или по ошибке укажет протокол HTTPS там, где в нем нет необходимости.

Атрибут requires-channel элемента <intercept-url> позволяет снять эту ответственность с программиста и переложить ее на конфигурацию Spring Security.

Рассмотрим в качестве примера форму регистрации в приложении Spitter. Приложение Spitter не просит указать номер кредитной карты, или номер карты социального обеспечения, или что-то жутко секретное, однако пользователи могут пожелать сохранить в тайне информацию о себе. Для этого необходимо настроить элемент <intercept-url>, описывающий доступ к адресу /spitter/form, как показано ниже:

```
<intercept-url pattern="/spitter/form" requires-channel="https"/>
```

Всякий раз, когда приложение будет получать запрос к адресу /spitter/form, фреймворк Spring Security будет обнаруживать требование https к протоколу и автоматически переадресовывать запрос на использование протокола HTTPS. Аналогично для доступа к главной странице приложения не требуется использовать протокол HTTPS, поэтому можно объявить, что доступ к ней всегда должен осуществляться по протоколу HTTP:

```
<intercept-url pattern="/home" requires-channel="http"/>
```

До настоящего момента демонстрировалось, как обезопасить веб-приложение на уровне запросов. При этом предполагалось, что основная задача системы безопасности состоит в том, чтобы помешать пользователю обращаться к определенным адресам URL, если он не имеет соответствующих привилегий. Но было бы неплохо также никогда не показывать ссылки тем пользователям, которые не смогут выполнить переход по ним. Посмотрим, что может предложить фреймворк Spring Security для безопасности представлений.

10.3. Безопасность на уровне представлений

Для обеспечения безопасности на уровне представлений в состав фреймворка Spring Security включена библиотека тегов JSP¹. Эта библиотека невелика и содержит всего три тега, которые перечислены в табл. 10.3.

Чтобы получить возможность использовать библиотеку тегов JSP, необходимо объявить ее в JSP-файле:

```
<%@ taglib prefix="security"
uri="http://www.springframework.org/security/tags" %>
```

После объявления библиотеку можно использовать. Рассмотрим поочередно все три тега JSP, входящие в состав Spring Security, и посмотрим, как они действуют.

¹ Для тех, кто предпочитает конструировать представления на основе фреймворка Velocity, в состав Spring Security входит коллекция макропределений Velocity, напоминающих соответствующие им теги JSP.

Таблица 10.3. Безопасность на уровне представлений поддерживается фреймворком Spring Security с помощью библиотеки тегов JSP

Тег JSP	Описание
<security:accesscontrollist>	Содержимое тега отображается, если текущий пользователь обладает одной из привилегий в указанном доменном объекте
<security:authentication>	Обеспечивает доступ к свойствам объекта аутентификации текущего пользователя
<security:authorize>	Содержимое тега отображается, если удовлетворяются указанные требования безопасности

10.3.1. Доступ к информации об аутентификации

Самое простое, на что способна библиотека тегов JSP, входящая в состав Spring Security, – предоставить доступ к информации об аутентификации пользователя. Например, для многих сайтов типично выводить приветствие в заголовке страницы, указывая в нем имя пользователя. Эту информацию можно получить с помощью тега <security:authentication>. Например:

```
Hello <security:authentication property="principal.username" />!
```

Атрибут *property* определяет свойство объекта аутентификации пользователя. Перечень доступных свойств зависит от того, как был аутентифицирован пользователь. Однако некоторые свойства, включая перечисленные в табл. 10.4, являются общими и доступны всегда.

В данном примере отображаемое свойство в действительности является вложенным свойством *username* свойства *principal*.

При таком использовании, как в примере выше, тег <security:authentication> отобразит значение указанного свойства в представлении. Но если потребуется лишь присвоить значение свойства переменной, тогда достаточно просто указать имя переменной в атрибуте *var*:

```
<security:authentication property="principal.username"
                           var="loginId"/>
```

Таблица 10.4. Тег <security:authentication> предоставляет доступ к информации аутентификации пользователя

Свойство	Описание
authorities	Коллекция объектов GrantedAuthority, представляющих привилегии, которыми обладает пользователь
credentials	Ключевая информация, использованная для проверки подлинности пользователя (часто это пароль)
details	Дополнительная информация об аутентификации (IP-адрес, серийный номер сертификата, идентификатор сеанса и т. д.)
principal	Основной объект с информацией о пользователе

По умолчанию переменная будет создана в области видимости страницы. Но с помощью атрибута scope можно также определить другую область видимости, такую как область видимости запроса или сеанса (или любую другую область видимости, доступную из javax.servlet.jsp.PageContext). Например, чтобы создать переменную в области видимости запроса, можно определить тег <security:authentication>, как показано ниже:

```
<security:authentication property="principal.username"
    var="loginId" scope="request" />
```

Тег <security:authentication> может пригодиться во многих ситуациях, но это лишь малая часть того, что может предоставить библиотека тегов JSP в Spring Security. Посмотрим, как обеспечить отображение содержимого страницы в зависимости от привилегий пользователя.

10.3.2. Отображение с учетом привилегий

Иногда некоторые фрагменты представления должны или не должны отображаться, в зависимости от привилегий пользователя. Бессмысленно отображать форму аутентификации, если пользователь уже аутентифицирован, или показывать персонализированное приветствие пользователю, который еще не аутентифицирован.

Тег <security:authorize> позволяет отображать фрагменты представлений в зависимости от привилегий, которыми обладает пользователь. Например, в приложении Spitter форма добавления нового сообщения не должна отображаться, если пользователь не обладает привилегией ROLE_SPITTER. Листинг 10.5 демонстрирует, как с помощью

тега <security:authorize> обеспечить отображение формы ввода сообщения, только если пользователь обладает привилегией ROLE_SPITTER.

Листинг 10.5. Отображение содержимого по условию с помощью тега <security:authorize>

```
<sec:authorize access="hasRole('ROLE_SPITTER')">      <!-- Только при обладании -->
                                                       <!-- привилегией ROLE_SPITTER -->
    <s:url value="/spittles" var="spittle_url" />
    <sf:form modelAttribute="spittle"
              action="${spittle_url}">

        <sf:label path="text"><s:message code="label.spittle"
                                         text="Enter spittle:"/></sf:label>

        <sf:textarea path="text" rows="2" cols="40" />
        <sf:errors path="text" />
        <br/>
        <div class="spitItSubmitIt">
            <input type="submit" value="Spit it!"
                  class="status-btn round-btn disabled" />
        </div>
    </sf:form>
</sec:authorize>
```

С атрибуте access указано выражение на языке SpEL, результат которого определяет, будет ли отображаться тело тега <security:authorize>. Здесь используется выражение hasRole('ROLE_SPITTER'), проверяющее, обладает ли текущий пользователь привилегией ROLE_SPITTER. Но точно так же в атрибуте access можно использовать любые выражения на языке SpEL, включая выражения, перечисленные в табл. 10.2 и имеющие непосредственное отношение к системе безопасности.

С помощью этих выражений можно реализовать весьма интересные ограничения. Например, представьте, что приложение включает некоторые административные функции, которые должны быть доступны лишь пользователю habuma. Для определения ограничения можно было бы использовать выражения isAuthenticated() и principal, как показано ниже:

```
<security:authorize
    access="isAuthenticated() and principal.username=='habuma'">
    <a href="/admin">Administration</a>
</security:authorize>
```

Уверен, что вы можете придумать еще более интересные выражения. Предоставляю вашему воображению возможность самому придумать еще более строгие ограничения. Благодаря поддержке SpEL количество вариантов практически бесконечно.

Но в примере, который я придумал, есть одна лазейка. Несмотря на возможность предоставить доступ к административным функциям только пользователю `habuma`, реализация такого ограничения с использованием выражения на языке SpEL далека от идеала. Разумеется, ссылка с адресом административной страницы не будет отображаться перед другими пользователями, но ничто не мешает им вручную ввести URL `/admin` в адресной строке браузера.

Вспомнив, о чем рассказывалось в этой главе, мы легко можем исправить данную проблему, достаточно лишь добавить новый элемент `<intercept-url>` в настройки безопасности, ограничивающий доступ к URL `/admin`:

```
<intercept-url pattern="/admin/**"
    access="hasRole('ROLE_ADMIN') and hasIpAddress('192.168.1.2')"/>
```

Теперь административные функции будут надежно заперты от посторонних. Сам URL будет доступен только одному пользователю, и ссылка с этим адресом не будет появляться перед пользователем, не обладающим соответствующими полномочиями. Но, чтобы добиться этого, пришлось объявить одно и то же выражение на языке SpEL в двух местах – в элементе `<intercept-url>` и в атрибуте `access` тега `<security:authorize>`. А возможно ли организовать отображение URL только при выполнении требований безопасности, определяемых для него?

Возможно, с помощью атрибута `url` тега `<security:authorize>`. В отличие от атрибута `access`, где ограничения определяются явно, атрибут `url` неявно ссылается на ограничения, накладываемые на шаблон URL. Поскольку ограничения безопасности для URL `/admin` уже определены в конфигурационном файле Spring Security, мы можем использовать атрибут `url`, как показано ниже:

```
<security:authorize url="/admin/**">
    <spring:url value="/admin" var="admin_url" />
    <br/><a href="#"> ${admin_url}>Admin</a>
</security:authorize>
```



Поскольку доступ к URL /admin может получить только аутентифицированный пользователь с привилегией ROLE_ADMIN и только если запрос приходит с определенного IP-адреса, тело тега <security:authorize> будет отображаться, только если выполняются все эти требования.

Другие атрибуты тега <security:authorize>. Помимо атрибутов access и url, тег <security:authorize> имеет еще три атрибута: ifAllGranted, ifAnyGranted и ifNotGranted. Эти атрибуты тега <security:authorize> позволяют обеспечить отображение информации в зависимости от наличия или отсутствия привилегий у пользователя.

До версии Spring Security 3.0 эти три атрибута были единственными, доступными в теге <security:authorize>. Но с введением поддержки языка выражений SpEL и атрибута access они потеряли актуальность. Их все еще можно использовать, но с помощью атрибута access можно реализовать то же самое и даже больше.

Теперь мы знаем, как реализовать различные ограничения на уровне представлений. Остается выяснить один вопрос: где хранится информация о пользователе? Иными словами: когда кто-то пытается аутентифицироваться в приложении, откуда Spring Security возьмет информацию для сравнения с данными аутентификации, представленными пользователем?

Все просто. Фреймворк Spring Security обладает достаточной гибкостью, чтобы использовать практически любое хранилище для хранения данных о пользователях. Рассмотрим некоторые из вариантов аутентификации, поддерживаемые фреймворком Spring Security.

10.4. Аутентификация пользователей

Каждое приложение имеет свои особенности. Эта истина особенно ярко проявляется в том, как каждое приложение хранит информацию о пользователях. Иногда для этого используется реляционная база данных. Иногда каталог LDAP. Некоторые приложения опираются на децентрализованные системы аутентификации пользователей. А некоторые могут использовать сразу несколько стратегий.

К счастью, фреймворк Spring Security обладает достаточной гибкостью и способен использовать практически любую стратегию аутентификации. Spring Security готов поддержать многие механизмы аутентификации, включая аутентификацию пользователей при использовании:

- репозитория в памяти (настраиваемого в контексте приложения Spring);
- репозитория на основе JDBC;
- репозитория на основе LDAP;
- децентрализованных систем идентификации OpenID;
- централизованной системы аутентификации (Central Authentication System, CAS);
- сертификатов X.509;
- провайдеров на основе JAAS.

Если ни один из этих вариантов не соответствует вашим потребностям, легко можно реализовать собственную стратегию аутентификации и внедрить ее.

Рассмотрим поближе некоторые из наиболее часто используемых способов аутентификации, поддерживаемых фреймворком Spring Security.

10.4.1. Настройка репозитория в памяти

Простейшим из имеющихся способов аутентификации является объявление информации о пользователях непосредственно в файле конфигурации Spring. Делается это за счет настройки службы учета пользователей с помощью элемента `<user-service>`, имеющегося в пространстве имен Spring Security:

```
<user-service id="userService">
    <user name="habuma" password="letmein"
          authorities="ROLE_SPITTER,ROLE_ADMIN"/>
    <user name="twoqubed" password="longhorns"
          authorities="ROLE_SPITTER"/>
    <user name="admin" password="admin"
          authorities="ROLE_ADMIN"/>
</user-service>
```

Функции службы учета пользователей фактически выполняет объект доступа к данным, который отыскивает информацию по указанному идентификатору. В случае с элементом `<user-service>` эта информация о пользователях объявляется внутри элемента `<user-service>`. Каждому пользователю, который может пользоваться приложением, соответствует отдельный элемент `<user>`. Атрибуты `name` и `password` определяют, соответственно, имя пользователя и пароль. А в атрибуте `authorities` указывается список привилегий, перечис-

ленных через запятую, которые определяют доступные пользователю операции.

Напомню, что выше (в разделе 10.2.3) в конфигурации Spring Security мы разрешили доступ ко всем URL только пользователям с привилегией `ROLE_SPITTER`. В данном случае эту привилегию получают пользователи `habuma` и `twoqubed`, а пользователь `admin` – нет.

Теперь служба учета пользователей готова к работе и ожидает возможности отыскать информацию для аутентификации. Осталось лишь внедрить ее в подсистему аутентификации Spring Security:

```
<authentication-manager>
    <authentication-provider user-service-ref="userService" />
</authentication-manager>
```

Элемент `<authentication-manager>` регистрирует компонент подсистемы аутентификации. Точнее, он регистрирует экземпляр `ProviderManager`, который делегирует функции аутентификации пользователей одной или более провайдерам аутентификации. В данном случае используется провайдер, опирающийся на службу учета пользователей. В приложении уже имеется служба учета пользователей. Поэтому осталось лишь внедрить ее, указав в атрибуте `user-service` элемента `<authentication-provider>`.

В этом примере провайдер аутентификации и служба учета пользователей были объявлены отдельно друг от друга, а затем связаны между собой. Однако существует также возможность встроить службу учета пользователей непосредственно в провайдер аутентификации:

```
<authentication-provider>
    <user-service id="userService">
        <user name="habuma" password="letmein"
            authorities="ROLE_SPITTER,ROLE_ADMIN"/>
        ...
    </user-service>
</authentication-provider>
```

Способ встраивания элемента `<user-service>` в элемент `<authentication-provider>` не имеет каких-либо существенных преимуществ, но для кого-то такой способ настройки может выглядеть более логичным.

Прием определения информации о пользователях в контексте приложения Spring удобно использовать во время тестирования,

когда вы только приступаете к настройке системы безопасности приложения. Но в условиях нормальной эксплуатации приложения этот способ никуда не годится. Чаще всего информация о пользователях сохраняется в базе данных или на сервере каталогов. Поэтому далее посмотрим, как зарегистрировать службу учета пользователей, которая ищет информацию в реляционной базе данных.

10.4.2. Аутентификация с использованием базы данных

Многие приложения хранят информацию о пользователях, включая имена и пароли, в реляционной базе данных. Если в приложении предполагается хранить информацию о пользователях именно таким способом, для этой цели с успехом можно использовать элемент Spring Security <jdbc-user-service>.

Элемент <jdbc-user-service> используется так же, как и элемент <user-service>, включая его внедрение в атрибут user-service элемента <authentication-provider> или встраивание внутрь элемента <authentication-provider>. Ниже демонстрируется пример настройки <jdbc-user-service>, где определяется значение атрибута id, благодаря чему элемент может быть объявлен отдельно и затем внедрен в элемент <authentication-provider>:

```
<jdbc-user-service id="userService"
    data-source-ref="dataSource" />
```

Для извлечения информации о пользователях из базы данных элемент <jdbc-user-service> использует источник данных JDBC, внедренный посредством атрибута data-source. Без дополнительных настроек служба учета пользователей будет извлекать информацию, используя следующий SQL-запрос:

```
select username, password, enabled
    from users
    where username = ?
```

И хотя сейчас мы ведем речь об аутентификации пользователя, определение привилегий пользователя тоже можно отнести к процедуре аутентификации. По умолчанию элемент <jdbc-user-service> будет использовать следующий SQL-запрос для определения привилегий пользователя с указанным именем:

```
select username,authority
  from authorities
 where username = ?
```

Все это хорошо, если так случилось, что информация о пользователях в базе данных приложения хранится в таблицах, соответствующих этим запросам. Но готов поспорить, что в большинстве приложений это не так. Если говорить о приложении Spitter, информация о пользователях хранится в таблице spitter. Очевидно, что настройки по умолчанию не подходят для данного случая.

Таблица 10.5. Атрибуты элемента <jdbc-user-service>, изменяющие SQL-запрос, который используется для извлечения информации о пользователе

Атрибут	Описание
users-by-username-query	Запрос имени пользователя, пароля и признака «разрешен/запрещен»
authorities-by-username-query	Запрос привилегий для указанного пользователя
group-authorities-by-username-query	Запрос группы привилегий для указанного пользователя

К счастью, элемент <jdbc-user-service> легко настроить на использование любых других запросов, лучше подходящих под нужды приложения. В табл. 10.5 перечислены атрибуты элемента <jdbc-user-service>, которые можно использовать для настройки его поведения.

Для приложения Spitter следует определить атрибуты users-by-username-query и authorities-by-username-query, как показано ниже:

```
<jdbc-user-service id="userService"
  data-source-ref="dataSource"
  users-by-username-query=
    "select username, password, true from spitter where username=?"
  authorities-by-username-query=
    "select username,'ROLE_SPITTER' from spitter where username=?" />
```

В приложении Spitter имена пользователей и пароли хранятся в таблице spitter, в полях username и password соответственно. Но мы не рассматривали идею возможности активации или деактивации учетных записей пользователей, предполагая, что все учетные записи активны. Поэтому SQL-запрос написан так, что в качестве

признака «разрешен/запрещен» он всегда возвращает `true` для всех пользователей.

Мы также не позабыли о распределении пользователей приложения Spitter по разным уровням привилегий. Все пользователи обладают одинаковыми привилегиями. В действительности в базе данных приложения Spitter отсутствует таблица, где хранится информация о привилегиях пользователей. Поэтому в атрибуте `authorities-by-username-query` указан запрос, возвращающий привилегию `ROLE_SPITTER` для любых пользователей.

Реляционные базы данных часто используются для хранения информации о пользователях, но не менее (если не более) часто для аутентификации применяется сервер каталогов LDAP. Поэтому посмотрим, как настроить Spring Security на использование LDAP в качестве хранилища информации о пользователях.

10.4.3. Аутентификация с использованием LDAP

Все мы один или два раза видели структуру какой-либо организации. Большинство организаций имеют иерархическую структуру. Служащие подчинены своими руководителями, руководители – директорам, директора – президентам и т. д. В подобной иерархии всегда можно усмотреть аналогии с иерархической организацией правил безопасности. Служащим отдела кадров, вероятно, потребуются привилегии, дающие право вести учет персонала. Руководителям наверняка потребуется более широкий круг привилегий, чем их подчиненным.

Реляционные базы данных с успехом могут использоваться для хранения информации, но они плохо подходят для представления данных с иерархической организацией. С другой стороны, каталоги LDAP превосходно справляются с этим. По этой причине очень часто для отражения структурной организации компании используется каталог LDAP, и также часто ограничения системы безопасности отображаются на записи в каталоге.

Чтобы использовать механизм аутентификации на основе LDAP, сначала необходимо задействовать модуль LDAP, входящий в состав Spring Security, и настроить аутентификацию посредством LDAP в контексте приложения Spring. Что касается настройки аутентификации через LDAP, имеются два варианта:

- с помощью провайдера аутентификации с поддержкой LDAP;
- с помощью службы учета пользователей с поддержкой LDAP.

По большому счету, это даже не выбор. Но есть некоторые нюансы, которые необходимо учитывать при выборе того или иного механизма.

Объявление провайдера с поддержкой LDAP

Настраивая службы учета пользователей с хранилищем в памяти и на основе JDBC, мы объявляли элемент `<authentication-provider>` и внедряли в него службу учета пользователей. То же самое можно сделать и со службой учета пользователей на основе LDAP (и чуть ниже будет показано, как это сделать). Но более простой способ заключается в использовании специализированного провайдера аутентификации, обладающего поддержкой LDAP, для чего необходимо объявить элемент `<ldap-authentication-provider>` в элемент `<authentication-manager>`:

```
<authentication-manager alias="authenticationManager">
    <ldap-authentication-provider
        user-search-filter="(uid={0})"
        group-search-filter="member={0}"/>
</authentication-manager>
```

Атрибуты `user-search-filter` и `group-search-filter` определяют фильтры для базовых запросов LDAP, которые используются для поиска пользователей и групп. По умолчанию базовые запросы на получение информации о пользователях и группах пусты, вследствие чего поиск будет выполняться от корня иерархии LDAP. Но есть возможность изменить базовые запросы:

```
<ldap-user-service id="userService"
    user-search-base="ou=people"
    user-search-filter="(uid={0})"
    group-search-base="ou=groups"
    group-search-filter="member={0}" />
```

Атрибут `user-search-base` определяет базовый запрос для поиска пользователей. Аналогично атрибут `group-search-base` определяет базовый запрос для поиска групп. Здесь указывается, что поиск пользователей должен выполняться не от корня иерархии, а там, где организационной единицей является человек. Аналогично по-

иск групп должен выполняться там, где организационной единицей являются группы.

Настройка сравнения паролей

По умолчанию процедура аутентификации с использованием LDAP заключается в выполнении операции связывания, когда аутентификация пользователя производится непосредственно на сервере LDAP. Другой вариант состоит в том, чтобы выполнить процедуру *сравнения*. Она заключается в отправке пароля в каталог LDAP и получении результатов сравнения с сервера. Поскольку сравнение выполняется сервером LDAP, секретность фактического пароля не нарушается.

Если предпочтение будет отдано аутентификации посредством сравнения паролей, реализовать ее можно, объявив элемент <password-compare>:

```
<ldap-authentication-provider
    user-search-filter="(uid={0})"
    group-search-filter="member={0}">
    <password-compare />
</ldap-authentication-provider>
```

Согласно этому объявлению, пароль, введенный пользователем в форме аутентификации, будет сравниваться со значением атрибута userPassword в учетной записи пользователя на сервере LDAP. Если пароль хранится в другом атрибуте, его имя можно определить в атрибуте password-attribute:

```
<password-compare hash="md5"
    password-attribute="passcode" />
```

Это здорово, что при использовании процедуры аутентификации посредством сравнения паролей секретность фактического пароля не нарушается. Но принятый от пользователя пароль необходимо передать серверу LDAP по сети, где он может быть перехвачен злоумышленником. Чтобы предотвратить такую возможность, можно определить стратегию *шифрования* трафика, присвоив атрибуту hash одно из следующих значений:

- {sha};
- {ssha};
- md4;

- md5;
- plaintext;
- sha;
- sha-256.

В нашем примере мы будем шифровать пароли с использованием алгоритма MD5, установив значение `md5` в атрибуте `hash`.

Обращение к удаленному серверу LDAP

Осталась еще одна нерешенная проблема – как указать фактический адрес сервера LDAP и местоположение данных. Мы успешно справились с настройкой Spring для аутентификации на сервере LDAP, но как быть с самим сервером?

По умолчанию модуль Spring Security LDAP предполагает, что сервер LDAP выполняется на локальном компьютере (`localhost`) и для приема запросов использует порт с номером 33389. Но если сервер LDAP выполняется на другом компьютере, для настройки его адреса можно воспользоваться элементом `<ldap-server>`:

```
<ldap-server url="ldap://habuma.com:389/dc=habuma,dc=com" />
```

Здесь адрес сервера LDAP определяется атрибутом `url`¹.

Настройка встроенного сервера LDAP

Если так случилось, что у вас нет сервера LDAP, готового выполнить запросы на аутентификацию, с помощью все того же элемента `<ldap-server>` можно настроить работу со встроенным сервером LDAP. Для этого достаточно просто убрать атрибут `url`. Например:

```
<ldap-server root="dc=habuma,dc=com" />
```

Атрибут `root` указывать необязательно. Но по умолчанию он имеет значение `dc=springframework,dc=org`, что, как мне кажется, не совсем то, что вам захочется использовать в качестве корня.

Когда сервер LDAP запустится, он попытается загрузить данные из файлов в формате LDIF, которые ему удастся обнаружить в библиотеке классов (`classpath`). Формат LDIF (LDAP Data Interchange Format – формат обмена данными с сервером LDAP) – это стандартный способ представления данных LDAP в виде текстовых файлов.

¹ Даже не пытайтесь использовать этот URL. Это всего лишь пример. По указанному URL в действительности нет никакого сервера LDAP.

Каждая запись занимает одну или более строк и содержит пары имя:значение. Друг от друга записи отделяются пустыми строками¹.

Те, кто предпочитает явно указывать, какой файл LDIF следует загрузить, могут воспользоваться атрибутом `ldif`:

```
<ldap-server root="dc=habuma,dc=com"
    ldif="classpath:users.ldif" />
```

Здесь явно указывается, что сервер LDAP должен загрузить свое содержимое из файла `users.ldif`, находящегося в корне библиотеки классов. Для наиболее любопытных в листинге 10.6 приводится содержимое используемого здесь файла LDIF.

Листинг 10.6. Пример файла LDIF, где хранится информация для сервера LDAP

```
dn: ou=groups,dc=habuma,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups

dn: ou=people,dc=habuma,dc=com
objectclass: top
objectclass: organizationalUnit
ou: people

dn: uid=habuma,ou=people,dc=habuma,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Craig Walls
sn: Walls
uid: habuma
userPassword: password

dn: uid=jsmith,ou=people,dc=habuma,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
```

¹ Более подробное описание формата LDIF можно найти по адресу: <http://tools.ietf.org/html/rfc2849> (не менее подробное описание на русском языке можно найти по адресу: <http://pro-ldap.ru/tr/zytrax/ch8/> – прим. перев.).

```
cn: John Smith
sn: Smith
uid: jsmith
userPassword: password

dn: cn=spitter,ou=groups,dc=habuma,dc=com
objectclass: top
objectclass: groupOfNames
cn: spitter
member: uid=habuma,ou=people,dc=habuma,dc=com
```

Независимо от того, как выполняется аутентификация – с помощью базы данных или сервера LDAP, для пользователя гораздо удобнее вообще не сталкиваться с процедурой аутентификации непосредственно. Поэтому далее посмотрим, как настроить фреймворк Spring Security, чтобы он запоминал пользователя и не заставлял его выполнять процедуру аутентификации при каждом посещении приложения.

10.4.4. Включение функции «запомнить меня»

Возможность аутентификации пользователей имеет большое значение для приложения. Но, с точки зрения пользователя, было бы неплохо, если бы приложение не всегда требовало вводить имя пользователя и пароль. Именно по этой причине многие сайты предлагают возможность «запомнить» пользователя после аутентификации, чтобы при последующих обращениях к приложению ему не приходилось повторно проходить эту процедуру.

Фреймворк Spring Security дает возможность легко и просто включить в приложение поддержку функции «запомнить меня». Для этого достаточно всего лишь добавить элемент <remember-me> в элемент <http>:

```
<http auto-config="true" use-expressions="true">
    ...
    <remember-me
        key="spitterKey"
        tokenValiditySeconds="2419200" />
</http>
```

Здесь демонстрируется включение поддержки функции «запомнить меня» с некоторыми дополнительными настройками. При ис-

пользовании элемента `<remember-me>` без атрибутов автоматическая аутентификация реализуется за счет сохранения в cookie специального маркера, который остается действительным в течение двух недель. Однако в данном случае указывается, что маркер должен оставаться действительным в течение четырех недель (2 419 200 секунд).

Маркер, сохраняемый в cookie, конструируется из имени пользователя, пароля, даты истечения срока хранения и секретного ключа. Вся эта информация шифруется с применением алгоритма MD5. По умолчанию секретный ключ имеет значение `SpringSecured`, но здесь ему явно было присвоено значение `spitterKey`, чтобы обосновать приложение Spitter.

Достаточно просто. Теперь, когда поддержка функции «запомнить меня» включена, необходимо обеспечить некоторый способ, с помощью которого пользователи могли бы потребовать запомнить их. Для этого в запрос следует включить параметр `_spring_security_remember_me`. С этим прекрасно справляется простой флагок в форме аутентификации:

```
<input id="remember_me" name="_spring_security_remember_me"
       type="checkbox"/>
<label for="remember_me" class="inline">Remember me</label>
```

До настоящего момента все внимание уделялось обеспечению безопасности на уровне веб-запросов. Поскольку фреймворк Spring Security чаще используется для обеспечения безопасности веб-приложений, многие забывают, что он способен обеспечивать безопасность на уровне вызовов методов. Поэтому перейдем к изучению поддержки безопасности в Spring Security на уровне методов.

10.5. Защита методов

Как уже упоминалось выше, безопасность – это аспектно-ориентированное понятие. Поэтому в основе обеспечения безопасности на уровне методов в Spring Security лежит Spring AOP. Но вам едва ли придется напрямую сталкиваться с аспектами Spring Security. Все детали использования AOP, связанные с обеспечением безопасности методов, скрыты в единственном элементе: `<global-method-security>`. Ниже демонстрируется типичный пример использования `<global-method-security>`.



```
<global-method-security secured-annotations="enabled" />
```

Он настраивает Spring Security на обеспечение безопасности методов, отмеченных собственной аннотацией Spring Security: `@Secured`. Это лишь один из четырех возможных способов защиты методов, поддерживаемых фреймворком Spring Security:

- с помощью аннотации `@Secured`;
- с помощью аннотации JSR-250: `@RolesAllowed`;
- с помощью аннотаций Spring, проверяющих условия до и после вызова метода;
- посредством сопоставления метода с одним или более множествами точек внедрения.

Рассмотрим каждый из способов по очереди.

10.5.1. Защита методов с помощью аннотации `@Secured`

Когда в атрибуте `secured-annotations` элемента `<global-method-security>` указывается значение `enabled`, создается множество точек внедрения, благодаря чему аспекты Spring Security получают возможность обернуть методы компонентов, отмеченные аннотацией `@Secured`. Например:

```
@Secured("ROLE_SPITTER")
public void addSpittle(Spittle spittle) {
    // ...
}
```

Аннотация `@Secured` принимает массив строк. Каждая строка определяет привилегию, которой должен обладать пользователь, чтобы вызвать аннотированный метод. В данном случае фреймворку Spring Security сообщается, что он не должен позволять вызывать метод `saveSpittle()`, если пользователь не обладает привилегией `ROLE_SPITTER`.

Если аннотации `@Secured` передается несколько значений, для вызова защищенного метода пользователь должен обладать хотя бы одной из указанных привилегий. Например, следующий пример использования аннотации `@Secured` требует, чтобы пользователь обладал привилегией `ROLE_SPITTER` или `ROLE_ADMIN`:

```
@Secured({"ROLE_SPITTER", "ROLE_ADMIN"})
public void addSpittle(Spittle spittle) {
    // ...
}
```

Если защищенный метод попытается вызвать неавторизованный пользователь или пользователь, не обладающий необходимыми привилегиями, аспект, обертывающий метод, возбудит одно из исключений, реализованных в Spring Security (возможно, подкласс AuthenticationException или AccessDeniedException). В конечном счете исключение должно быть обработано приложением. Если вызов защищенного метода произойдет в процессе обработки веб-запроса, исключение будет автоматически обработано фильтрами Spring Security. Во всех остальных случаях вы должны будете написать собственный обработчик исключения.

Единственный недостаток аннотации @Secured – в том, что она является аннотацией Spring. Те, кто предпочитает использовать стандартные аннотации, могут использовать аннотацию @RolesAllowed.

10.5.2. Использование аннотации JSR-250 @RolesAllowed

Аннотация @RolesAllowed является практически полным эквивалентом аннотации @Secured. Единственное существенное отличие – в том, что аннотация @RolesAllowed является одной из стандартных аннотаций Java и определена в спецификации JSR-250¹.

Эти различия в большей степени относятся к политической стороне дела, чем к технической. Перевес в пользу стандартной аннотации @RolesAllowed может быть обусловлен необходимостью использования программного кода в контексте других фреймворков или API, обрабатывающих аннотацию.

Как бы то ни было, в случае выбора аннотации @RolesAllowed необходимо включить ее поддержку, определив значение enabled в атрибуте jsr250-annotations элемента <global-method-security>:

```
<global-method-security jsr250-annotations="enabled" />
```

¹ <http://jcp.org/en/jsr/summary?id=250>.

Хотя в данном примере был определен только атрибут `jsr250-annotations`, следует отметить, что он не исключает возможности определения атрибута `secured-annotations`. Эти два атрибута могут указываться одновременно. Более того, их можно даже использовать совместно с аннотациями фреймворка Spring, выполняемыми до и после вызова метода, о которых рассказывается далее.

10.5.3. Защита с помощью аннотаций, выполняемых до и после вызова

Аннотации `@Secured` и `@RolesAllowed` позволяют решить поставленную задачу, предотвращая возможность вызова методов неавторизованными пользователями, но это все, на что они способны. Иногда бывает необходимо реализовать более интересные ограничения, основанные не только на определении наличия некоторых привилегий у пользователя.

В Spring Security 3.0 появилось несколько новых аннотаций, позволяющих использовать выражения на языке SpEL для реализации более сложных ограничений на доступ к методам. Эти новые аннотации перечислены в табл. 10.6.

Таблица 10.6. Spring Security 3.0 предлагает четыре новые аннотации для защиты методов с применением выражений на языке SpEL

Аннотация	Описание
<code>@PreAuthorize</code>	Ограничивает доступ к методам перед их вызовом, опираясь на результат вычисления выражения
<code>@PostAuthorize</code>	Позволяет вызывать методы, но возбуждает исключение, если выражение возвращает значение <code>false</code>
<code>@PostFilter</code>	Позволяет вызывать методы, но фильтрует его результаты в соответствии со значением выражения
<code>@PreFilter</code>	Позволяет вызывать методы, но фильтрует входные данные перед фактическим вызовом метода

Примеры использования каждой из них будут представлены чуть ниже. Но прежде чем использовать, их поддержку необходимо включить в элементе `<global-method-security>`, указав значение `enabled` в атрибуте `pre-post-annotations`:

```
<global-method-security pre-post-annotations="enabled" />
```

После включения поддержки описываемых аннотаций можно приступать к их использованию для защиты методов. Начнем с аннотации `@PreAuthorize`.

Проверка условия перед вызовом метода

На первый взгляд может показаться, что аннотация `@PreAuthorize` является всего лишь эквивалентом аннотаций `@Secured` и `@RolesAllowed`, обладающим поддержкой выражений на языке SpEL. Фактически аннотацию `@PreAuthorize` можно использовать для ограничения доступа на основе привилегий, которыми обладает авторизованный пользователь:

```
@PreAuthorize("hasRole('ROLE_SPITTER')")
public void addSpittle(Spittle spittle) {
    // ...
}
```

Аннотация `@PreAuthorize` принимает строковый аргумент с выражением на языке SpEL. В примере выше используется функция `hasRole()`, предоставляемая фреймворком Spring Security, разрешающая доступ к методу, только если авторизованный пользователь обладает привилегией `ROLE_SPITTER`.

Однако на языке SpEL можно выразить гораздо более сложные требования. Например, представьте, что необходимо ограничить длину сообщений, посылаемых обычным пользователем, 140 символами и не применять это ограничение к привилегированному пользователю. Аннотации `@Secured` и `@RolesAllowed` оказались бы бесполезны в этом случае, а аннотация `@PreAuthorize` позволяет легко справиться с данной задачей:

```
@PreAuthorize("(hasRole('ROLE_SPITTER') and #spittle.text.length() <= 140)
               or hasRole('ROLE_PREMIUM'))"
public void addSpittle(Spittle spittle) {
    // ...
}
```

Фрагмент `#spittle` выражения является непосредственной ссылкой на параметр метода с тем же именем. Эта возможность позволяет фреймворку Spring Security исследовать параметры методов и использовать их в выражениях при принятии решения. Здесь проверяется длина свойства `text` объекта `Spitter`, чтобы убедиться, что

она не превышает допустимого значения, если метод вызывается обычным пользователем. А если пользователь является привилегированным пользователем, длина сообщения не имеет значения.

Проверка условия после вызова метода

Менее очевидный способ защиты методов заключается в проверке условия после вызова метода. Обычно в этом случае решение принимается на основе объекта, возвращаемого защищенным методом. Разумеется, это означает, что метод должен быть вызван и вернуть некоторое значение.

Помимо момента принятия решения, аннотация `@PostAuthorize` действует практически так же, как и аннотация `@PreAuthorize`. Например, предположим, что необходимо защитить метод `getSpittleById()`, позволив возвращать объект `Spittle`, только если он принадлежит авторизованному пользователю. Для этого можно было бы снабдить метод `getSpittleById()` аннотацией `@PostAuthorize`, как показано ниже:

```
@PostAuthorize("returnObject.spitter.username == principal.username")
public Spittle getSpittleById(long id) {
    // ...
}
```

Чтобы упростить доступ к объекту, возвращаемому защищенным методом, фреймворк Spring Security предоставляет имя `returnObject` в языке SpEL. В данном случае известно, что метод возвращает объект типа `Spittle`, поэтому выражение обращается к его свойству `spitter` и извлекает из него значение его свойства `username`.

С другой стороны оператора сравнения (`==`) используется встроенный объект `principal`, из которого извлекается значение свойства `username`. Объект `principal` – это еще одно специальное встроенное имя, предоставляемое фреймворком Spring Security, которое ссылается на главный объект, представляющий текущего авторизованного пользователя.

Если объект `Spittle` ссылается на объект `Spitter`, значение свойства `username` которого совпадает со значением свойства `username` объекта `principal`, то объект `Spittle` будет возвращен вызывающей программе. Иначе будет возбуждено исключение `AccessDeniedException` и вызывающая программа не получит объекта `Spittle`.

Имейте в виду, что, в отличие от методов, снабженных аннотацией `@PreAuthorize`, методы с аннотацией `@PostAuthorize` будут сначала выполнены, и только потом будет произведена проверка. Это означает,

что перед применением аннотации `@PostAuthorize` необходимо убедиться в отсутствии побочных эффектов в методе, которые могут привести к нарушению системы безопасности.

Фильтрация после вызова метода

Иногда бывает необходимо защитить не сам метод, а данные, возвращаемые этим методом. Например, представьте, что необходимо вернуть список сообщений пользователю, но при этом список должен содержать только сообщения, которые пользователь может удалить. В этом случае можно было бы аннотировать метод, как показано ниже:

```
@PreAuthorize("hasRole('ROLE_SPITTER')")
@PostFilter("filterObject.spitter.username == principal.name")
public List<Spittle> getABunchOfSpittles() {
    ...
}
```

Здесь аннотация `@PreAuthorize` позволяет вызывать метод лишь пользователям с привилегией `ROLE_SPITTER`. Если пользователь пройдет эту проверку, метод будет вызван и вернет список сообщений. Но аннотация `@PostFilter` отфильтрует этот список, оставив в нем только сообщения (объекты `Spittle`), принадлежащие пользователю.

Имя `filterObject` в выражении ссылается на отдельные элементы в списке (которые, как мы знаем, являются объектами `Spittle`), возвращаемом методом. Если свойство `username` свойства `spitter` в этом объекте совпадает с именем авторизованного пользователя (`principal.name` в выражении), тогда элемент остается в отфильтрованном списке. Иначе он удаляется.

Я знаю, что вы сейчас подумали. Можно было бы написать запрос, возвращающий только сообщения текущего пользователя. Такое решение можно было бы использовать, если бы правила позволяли пользователям удалять лишь свои сообщения.

А теперь усложним задачу и предположим, что помимо собственных сообщений пользователь должен обладать возможностью удалять сообщения, содержащие ненормативную лексику. Для этого перепишем выражение в аннотации `@PostFilter`, как показано ниже:

```
@PreAuthorize("hasRole('ROLE_SPITTER')")
@PostFilter("hasPermission(filterObject, 'delete')")
public List<Spittle> getSpittlesToDelete() {
    ...
}
```

В данном случае функция `hasPermission()` должна вернуть `true`, если пользователь обладает разрешением `delete` на удаление сообщения, на которое ссылается имя `filterObject`. Я сказал, что она должна вернуть `true`, но в действительности `hasPermission()` по умолчанию всегда возвращает `false`.

Если `hasPermission()` всегда по умолчанию возвращает `false`, тогда какой смысл использовать эту функцию? Вся прелест поведения по умолчанию – в том, что его всегда можно переопределить. Чтобы переопределить поведение функции `hasPermission()`, необходимо создать и зарегистрировать обработчик разрешений. Этую функцию берет на себя класс `SpittlePermissionEvaluator`, представленный в листинге 10.7.

Листинг 10.7. Обработчик разрешений, стоящий позади функции `hasPermission()`

```
package com.habuma.spitter.security;
import java.io.Serializable;
import org.springframework.security.access.PermissionEvaluator;
import org.springframework.security.core.Authentication;
import com.habuma.spitter.domain.Spittle;

public class SpittlePermissionEvaluator implements PermissionEvaluator {
    public boolean hasPermission(Authentication authentication,
        Object target, Object permission) {
        if (target instanceof Spittle) {
            Spittle spittle = (Spittle) target;
            if ("delete".equals(permission)) {
                return spittle.getSpitter().getUsername().equals(
                    authentication.getName()) || hasProfanity(spittle);
            }
        }
        throw new UnsupportedOperationException(
            "hasPermission not supported for object <" + target
            + "> and permission <" + permission + ">");
    }

    public boolean hasPermission(Authentication authentication,
        Serializable targetId, String targetType, Object permission) {
        throw new UnsupportedOperationException();
    }

    private boolean hasProfanity(Spittle spittle) {
        ...
        return false;
    }
}
```

Класс SpittlePermissionEvaluator реализует интерфейс PermissionEvaluator, требующий реализации двух различных методов hasPermission(). Один метод hasPermission() принимает два объекта Object. Первый – который требуется оценить, и второй – который определяет искомое разрешение. Второй метод hasPermission() может пригодиться, только когда доступен идентификатор оцениваемого объекта, который передается во втором параметре в виде объекта Serializable.

В данной ситуации предположим, что в оценке разрешений всегда будут участвовать только объекты Spittle, поэтому второй метод просто возбуждает исключение UnsupportedOperationException.

Что касается первого метода hasPermission(), он проверяет, является ли оцениваемый объект экземпляром класса Spittle, а объект искомого разрешения является разрешением на удаление. Если эти условия соблюдаются, метод сравнивает имя пользователя, создавшего сообщение, с именем текущего авторизованного пользователя, и проверяет наличие ненормативной лексики в тексте сообщения, передавая его методу hasProfanity()¹.

После реализации обработчика разрешений необходимо зарегистрировать его в Spring Security, чтобы обеспечить поддержку функции hasPermission() в выражении, указанном в аннотации @PostFilter. Поэтому необходимо создать компонент обработчика выражений и зарегистрировать его с помощью элемента <global-method-security>.

Для этого создадим компонент типа DefaultMethodSecurityExpressionHandler и внедрим в его свойство permissionEvaluator экземпляр нашего класса SpittlePermissionEvaluator:

```
<beans:bean id="expressionHandler" class=
    "org.springframework.security.access.expression.method.
     →DefaultMethodSecurityExpressionHandler">
    <beans:property name="permissionEvaluator">
        <beans:bean class=
            "com.habuma.spitter.security.SpittlePermissionEvaluator" />
    </beans:property>
</beans:bean>
```

Затем можно приступать к настройке компонента expressionHandler в элементе <global-method-security>, как показано ниже:

¹ Оставляю реализацию метода hasProfanity() на усмотрение читателей.



```
<global-method-security pre-post-annotations="enabled">
    <expression-handler ref="expressionHandler"/>
</global-method-security>
```

Прежде, при настройке элемента `<global-method-security>`, мы не указывали обработчика выражений. Но теперь мы заменили обработчик по умолчанию собственной реализацией, которая способна оценивать наличие разрешения.

10.5.4. Объявление точек внедрения для защиты методов

Ограничения безопасности, накладываемые на различные методы, часто зависят от конкретного метода. Поэтому прием, основанный на аннотировании каждого метода в отдельности, лучше соответствует предъявляемым требованиям. Но иногда имеет смысл применить одни и те же ограничения к нескольким методам.

Ограничить доступ к множеству методов можно с помощью элемента `<protect-pointcut>`, поместив его в элемент `<global-method-security>`. Например:

```
<global-method-security>
    <protect-pointcut access="ROLE_SPITTER"
        expression=
            "execution(@com.habuma.spitter.Sensitive * *.*(String))"/>
</global-method-security>
```

В атрибуте `expression` указывается выражение AspectJ, определяющее множество точек внедрения. В данном случае выражение идентифицирует все методы, аннотированные нестандартной аннотацией `@Sensitive`. При этом атрибут `access` определяет, какими привилегиями должен обладать авторизованный пользователь, чтобы получить доступ к методам, идентифицируемым атрибутом `expression`.

10.6. В заключение

Безопасность – критически важный аспект многих приложений. Фреймворк Spring Security предоставляет простые, гибкие и мощные механизмы, позволяющие обезопасить приложение.

С помощью последовательности сервлет-фильтров Spring Security может контролировать доступ к веб-ресурсам, включая контроллеры

Spring MVC. А с помощью аспектов Spring Security можно организовать защиту методов. Благодаря наличию в Spring Security конфигурационного пространства имен вам не придется непосредственно сталкиваться с фильтрами или аспектами, а настройка безопасности получится краткой и выразительной.

Что касается аутентификации пользователей, Spring Security предлагает несколько вариантов. Выше было показано, как настраивать аутентификацию с использованием хранилища информации о пользователях в памяти, в реляционной базе данных и на сервере каталогов LDAP.

Далее будут рассматриваться способы интеграции приложений на основе фреймворка Spring с другими приложениями. Начиная со следующей главы, мы приступим к изучению поддержки в Spring возможности удаленных взаимодействий, включая RMI и веб-службы.



Часть III. Интеграция Spring

В частях 1 и 2 вы познакомились с основами фреймворка Spring и с примерами разработки приложений с использованием поддержки хранилищ и транзакций в Spring и веб-фреймворков. В третьей части рассказывается, как сделать еще один шаг вперед и интегрировать свои приложения с другими службами и приложениями.

В главе 11 «Взаимодействие с удаленными службами» вы узнаете, как обеспечить доступ к объектам приложения как к удаленным службам. Вы также узнаете, как реализовать прозрачный доступ к удаленным службам, как если бы они были объектами в приложении. Мы исследуем технологии удаленных взаимодействий, включая RMI, Hessian/Burlap, Spring HTTP Invoker и веб-службы с поддержкой JAX-RPC и JAX-WS.

В противоположность RPC-подобным службам, представленным в главе 11, в главе 12 исследуются приемы интеграции с ресурсами REST с применением Spring MVC.

В главе 13 «Обмен сообщениями в Spring» рассматривается иной подход к интеграции приложений, где демонстрируется возможность использования Spring совместно с JMS для организации асинхронного обмена сообщениями между приложениями.

Мониторинг компонентов Spring и управление ими является темой главы 14 «Управление компонентами Spring с помощью JMX». В этой главе вы узнаете, как автоматически представлять компоненты, настроенные в Spring, в виде компонентов JMX MBeans.

Глава 15 «Создание веб-служб на основе модели contract-first» рассматривает иной подход к созданию веб-служб и демонстрирует, как можно использовать фреймворк Spring Web Services для создания веб-служб на основе соглашений.

Глава 16 «Spring и Enterprise JavaBeans» охватывает вопросы организации взаимодействий между Spring и компонентами EJB. Здесь рассказывается, как внедрять компоненты EJB в приложения на основе Spring и как создавать сеансовые компоненты EJB с под-

держкой Spring. В этой главе мы также коротко познакомимся с особенностями разработки компонентов EJB 3 в Spring.

В завершение книги, в главе 17 «Прочее», охватываются некоторые темы, достаточно важные, чтобы обсудить их, но недостаточно объемные, чтобы выделить их в отдельные главы. В этой главе вы узнаете, как импортировать внешние настройки, связывать JNDI-ресурсы как компоненты Spring, отправлять электронные письма, выполнять задания по расписанию и объявлять методы, которые должны выполняться асинхронно, подобно фоновым заданиям.



Глава 11. Взаимодействие с удаленными службами

В этой главе рассматриваются следующие темы:

- ❑ доступ к службам RMI и их реализация;
- ❑ использование служб Hessian и Burlap;
- ❑ применение Spring HTTP Invoker;
- ❑ использование Spring для взаимодействия с веб-службами.

Представьте на мгновение, что вы оказались на пустынном острове. Кому-то это может показаться несбыточной мечтой. В конце концов, кто не хотел бы провести некоторое время в одиночку, пребывая в блаженном неведении о том, что творится в окружающем мире?

Но одиночество не может доставлять удовольствие вечно. Даже если покойное уединение приносит вам радость, это будет длиться до тех пор, пока вы не проголодаетесь и не соскучитесь по общению с другими людьми. На кокосовых орехах и пойманной рыбе можно жить довольно долго. Но рано или поздно вам потребуются другая пища, чистая одежда и другие вещи. И если вам не удастся быстро войти в контакт с другими людьми, вы можете закончить тем, что начнете разговаривать с волейбольным мячом!

Многие приложения напоминают подобных путешественников, волею судьбы оказавшихся на необитаемом острове. Независимые и автономные на первый взгляд, они в действительности могут тесно взаимодействовать с другими системами, как находящимися внутри вашей организации, так и за ее пределами.

Например, представьте систему управления закупками оборудования, которая должна взаимодействовать с системой управления поставками в компании-производителе. Возможно, система управления кадрами в вашей компании должна каким-то образом взаимодействовать с системой начисления заработной платы. Или системе начисления заработной платы необходимо взаимодействовать с внешней системой, которая печатает, отправляемые по почте чеки

с зарплатой. Одним словом, приложению необходимо взаимодействовать с другими системами и удаленными службами.

Для Java-разработчиков доступны несколько технологий удаленных взаимодействий, включая:

- ❑ вызов удаленных методов (Remote Method Invocation, RMI);
- ❑ Cauchy Hessian и Burlap;
- ❑ механизмы удаленных взаимодействий по протоколу HTTP в Spring;
- ❑ веб-службы на основе JAX-RPC и JAX-WS.

Фреймворк Spring предлагает на выбор достаточно широкий спектр технологий создания удаленных служб и взаимодействия с ними. В этой главе вы узнаете, насколько фреймворк Spring упрощает и дополняет эти службы. Но сначала заложим фундамент для дальнейшего обсуждения обзором механизмов удаленных взаимодействий, имеющихся в Spring.

11.1. Обзор механизмов удаленных взаимодействий в Spring

Удаленное взаимодействие – это диалог между клиентским приложением и службой. На стороне клиента востребованы некоторые функциональные возможности, не входящие в компетенцию приложения. Поэтому приложение обращается к другой системе, способной предоставить необходимую функциональность. Удаленное приложение обеспечивает доступ к своей функциональности посредством *удаленной службы*.

Представьте, что нам захотелось обеспечить доступ к некоторым функциям приложения Spitter как к удаленным службам для использования другими приложениями. Например, обычными или мобильными приложениями, реализующими интерфейс к приложению Spitter, в дополнение к веб-интерфейсу, как показано на рис. 11.1. Для этого необходимо предоставить доступ к базовым функциям интерфейса `SpitterService`, реализованного в виде удаленной службы.

Диалог между другими приложениями и приложением Spitter начинается с *вызыва удаленной процедуры* (Remote Procedure Call, RPC) в клиентском приложении. На первый взгляд, RPC напоминает вызов метода локального объекта. Оба вызова являются синхронными операциями и блокируют дальнейшее выполнение программы, пока вызов процедуры не завершится.



Рис. 11.1. Сторонний клиент может взаимодействовать с приложением Spitter, выполняя удаленные вызовы экспортированных методов

Здесь можно провести аналогию с общением между двумя людьми. Если вы обсуждаете на работе итоги футбольного матча, состоявшегося в выходные, следовательно, вы участвуете в локальном диалоге – диалоге между двумя людьми, находящимися в одной комнате. Аналогично вызов локального метода является переходом потока выполнения из одного блока кода в другой внутри одного приложения.

С другой стороны, если вы поднимаете телефонную трубку, чтобы позвонить клиенту в другой город, вы будете участвовать в удаленном диалоге через сеть телефонных проводов. Аналогично, когда выполняется вызов удаленной процедуры, управление выполнением передается от одного приложения другому, теоретически выполняющемуся на другом компьютере, через компьютерную сеть.

Фреймворк Spring поддерживает несколько моделей RPC, включая *вызов удаленных методов* (Remote Method Invocation, RMI), Caucho Hessian и Burlap, а также Spring HTTP Invoker. Все эти модели перечислены в табл. 11.1, где также кратко описываются ситуации, где они могут пригодиться.

Независимо от выбранной модели удаленных взаимодействий все проблемы их использования упираются в поддержку их фреймворком Spring. Это означает, что, научившись настраивать Spring для работы с одной моделью, вам почти не придется изучать ничего нового, чтобы задействовать другую модель.

При использовании любой модели, службы в приложении можно настраивать как компоненты, управляемые фреймворком Spring. Это достигается за счет применения фабричного компонента, создающего прокси-объекты, позволяющие связывать удаленные службы со свойствами компонентов, как если бы они были локальными объектами. Как это делается, показано на рис. 11.2.

Таблица 11.1. Spring поддерживает RPC посредством нескольких технологий удаленных взаимодействий

Модель RPC	Может использоваться для...
Вызов удаленных методов (Remote Method Invocation, RMI)	Доступа к службам, реализованным на языке Java, а также для реализации таких служб, когда отсутствуют сетевые ограничения, такие как брандмауэры
Hessian или Burlap	Доступа к службам, реализованным на языке Java, посредством протокола HTTP, а также для реализации таких служб, когда отсутствуют сетевые ограничения, такие как брандмауэры
HTTP Invoker	Доступа к службам, реализованным на основе фреймворка Spring, а также для реализации таких служб, когда имеются какие-либо сетевые ограничения и когда желательно использовать механизмы сериализации Java в формат XML или другие частные форматы
JAX-RPC и JAX-WS	Доступа к платформонезависимым веб-службам посредством протокола SOAP, а также для реализации таких служб

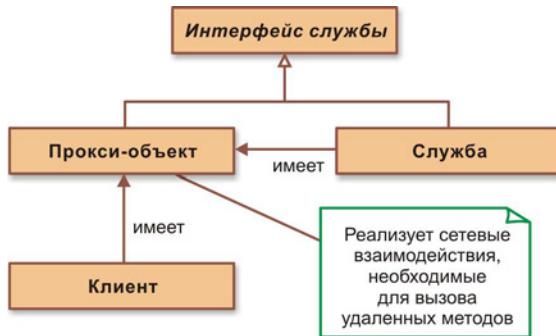


Рис. 11.2. В Spring обращение к удаленным службам выполняется через прокси-объекты, ссылки на которые можно внедрить в программный код клиента, как если бы эти службы были обычными компонентами Spring

Клиент обращается к прокси-объекту, как если бы он реализовал функциональность службы. Прокси-объект взаимодействует с удаленной службой от имени клиента. Он реализует все операции по установлению соединения и выполнению вызовов методов удаленной службы.

К тому же, если результатом обращения к удаленной службе является исключение `java.rmi.RemoteException`, прокси-объект обра-

ботает его и возбудит неконтролируемое исключение `RemoteAccessException`. Удаленные исключения обычно сигнализируют о проблемах с сетью или с настройками, которые сложно обработать на стороне клиента. Поскольку в распоряжении клиента не так много вариантов обработки удаленных исключений, возбуждение исключения `RemoteAccessException` на стороне клиента не является обязательным.

На стороне службы имеется возможность экспортить функциональность любого компонента с использованием любой модели из перечисленных в табл. 11.1. На рис. 11.3 показано, как объект-экспортер экспортирует методы компонента в виде API удаленной службы.

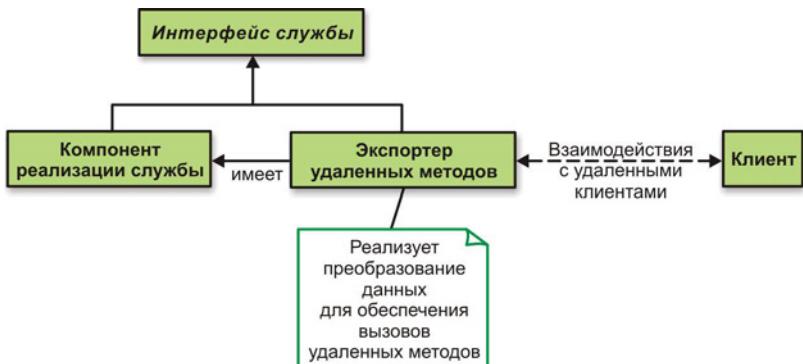


Рис. 11.3. Компоненты, выполняющиеся под управлением Spring, могут экспортirоваться как удаленные службы с использованием шлюзов удаленных взаимодействий

И реализация взаимодействий с удаленными службами, и реализация самих удаленных служб в Spring в значительной степени является проблемой конфигурирования. Вам не придется писать программный код, реализующий удаленные взаимодействия. При реализации компонентов вам не придется беспокоиться о поддержке RPC (однако все компоненты, передаваемые в удаленные вызовы или возвращаемые ими, в обязательном порядке должны реализовать интерфейс `java.io.Serializable`).

Начнем с исследования поддержки в Spring удаленных взаимодействий в модели RMI – оригинальной для Java технологии удаленных взаимодействий.

11.2. Использование RMI

Имеющие достаточно продолжительный опыт разработки приложений на языке Java наверняка слышали (и, возможно, использовали) механизм вызова удаленных методов (Remote Method Invocation, RMI). Поддержка RMI, впервые появившаяся в JDK 1.1, обеспечивает широкие возможности удаленных взаимодействий между программами на языке Java. До появления поддержки RMI Java-программистам был доступен единственный механизм удаленных взаимодействий – CORBA (требовавший приобретения стороннего брокера объектных запросов (Object Request Broker, ORB)), иначе приходилось опускаться на низкий уровень и заниматься программированием сокетов.

Однако и с применением механизма RMI реализация доступа к удаленным службам весьма утомительна и требует вручную писать немалый объем кода. Фреймворк Spring упрощает модель RMI, предоставляя фабричный компонент, создающий прокси-объекты, который позволяет внедрять службы RMI в приложения на основе Spring, как если бы они были локальными компонентами JavaBeans. Spring также предоставляет объект-экспортер, упрощающий преобразование компонентов, управляемых фреймворком Spring, в службы RMI.

На примере приложения Spitter я покажу, как внедрить службу RMI в контекст клиентского приложения на основе Spring. Но сначала посмотрим, как можно использовать объект-экспортер для экспортирования реализации компонента SpitterService в виде службы RMI.

11.2.1. Экспортирование службы RMI

Если прежде вам приходилось создавать службы RMI, вы уже знаете, что этот процесс состоит из нескольких этапов:

1. Написать класс, реализующий службу, с методами, возбуждающими исключение `java.rmi.RemoteException`.
2. Создать интерфейс службы, расширяющий `java.rmi.Remote`.
3. Запустить RMI-компилятор (`rmic`), чтобы создать классы-заготовки для реализации клиента и сервера.
4. Запустить RMI-реестр, где будет выполняться служба.
5. Зарегистрировать службу в RMI-реестре.

М-да! Слишком много, чтобы организовать работу простенькой службы RMI. Но хуже всего, что тут и там могут возбуждаться исключения `RemoteExceptions` и `MalformedURLExceptions`. Обычно эти исключения свидетельствуют о фатальных ошибках, обработка кото-

рых не имеет большого смысла, но вы вынуждены будете писать массу шаблонного кода, перехватывающего и обрабатывающего их даже при том, что это практически бессмысленно.

Очевидно, что организация службы RMI связана с большим объемом ручной работы. А может фреймворк Spring как-то исправить эту ситуацию?

Настройка службы RMI в Spring

К счастью, фреймворк Spring предоставляет простой способ публикации служб RMI. Вместо того чтобы писать классы, учитывающие особенности RMI, с методами, возбуждающими исключение `RemoteException`, достаточно написать POJO, реализующий функциональность службы, а все остальное Spring возьмет на себя.

Служба RMI, которая будет создана далее, экспортирует методы интерфейса `SpitterService`. В качестве напоминания в листинге 11.1 приводится объявление этого интерфейса.

Листинг 11.1. Интерфейс SpitterService определяет службу для приложения Spitter

```
package com.habuma.spitter.service;

import java.util.List;

import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;

public interface SpitterService {
    List<Spittle> getRecentSpittles(int count);
    void saveSpittle(Spittle spittle);

    void saveSpitter(Spitter spitter);
    Spitter getSpitter(long id);
    void startFollowing(Spitter follower, Spitter followee);

    List<Spittle> getSpittlesForSpitter(Spitter spitter);
    List<Spittle> getSpittlesForSpitter(String username);
    Spitter getSpitter(String username);

    Spittle getSpittleById(long id);
    void deleteSpittle(long id);

    List<Spitter> getAllSpitters();
}
```

При традиционном подходе к реализации службы RMI все эти методы в `SpitterService` и в `SpitterServiceImpl` должны были бы быть объявлены как возбуждающие исключение `java.rmi.RemoteException`. Но в данном случае предполагается, что подобный интерфейс и его реализация будут превращены в службу RMI с использованием компонента `RmiServiceExporter`, входящего в состав Spring, поэтому существующей реализации вполне достаточно.

Компонент `RmiServiceExporter` способен экспортить любые компоненты Spring как службы RMI. Как отмечено на рис. 11.4, компонент `RmiServiceExporter` обертывает требуемый компонент классом адаптера. Затем класс-адаптер заносится в реестр RMI и выполняет передачу запросов классу службы, в данном случае классу `SpitterServiceImpl`.

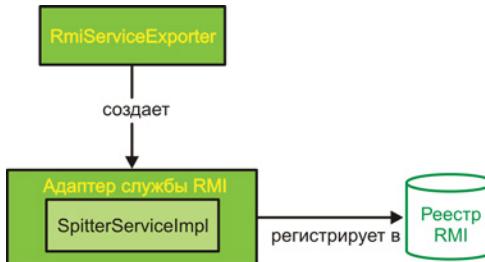


Рис. 11.4. Компонент `RmiServiceExporter` превращает POJO в службу RMI, обертывая его адаптером службы и регистрируя адаптер в реестре RMI

Самый простой способ задействовать `RmiServiceExporter` для экспортации `SpitterServiceImpl` в виде службы RMI состоит в том, чтобы выполнить соответствующие настройки в конфигурационном XML-файле Spring, как показано ниже:

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
    <p:service-ref>spitterService</p:service-ref>
    <p:serviceName>"SpitterService"</p:serviceName>
    <p:serviceInterface>"com.habuma.spitter.service.SpitterService" />

```

Здесь компонент `spitterService` внедряется в свойство `service`, чтобы сообщить компоненту `RmiServiceExporter`, что он должен экспорттировать данный компонент как службу RMI. Свойство `serviceName` определяет имя службы RMI. А свойство `serviceInterface` – интерфейс, который реализует служба.

По умолчанию компонент `RmiServiceExporter` попытается соединиться с реестром RMI, действующим на локальном компьютере и принимающим запросы на порту с номером 1099. Если на этом порту реестр RMI не будет обнаружен, `RmiServiceExporter` попытается запустить его. Если необходимо установить связь с реестром RMI на другом порту и/или на другом компьютере, параметры соединения можно настроить с помощью свойств `registryPort` и `registryHost`. Например, при следующих настройках компонент `RmiServiceExporter` попытается установить соединение с реестром RMI, действующим на компьютере `rmi.spitter.com` и порту с номером 1199:

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter"
      p:service-ref="spitterService"
      p:serviceName="SpitterService"
      p:serviceInterface="com.habuma.spitter.service.SpitterService"
      p:registryHost="rmi.spitter.com"
      p:registryPort="1199"/>
```

Это все настройки, необходимые Spring, чтобы превратить компонент в службу RMI. Теперь, когда экспортование службы выполнено, можно приступить к созданию альтернативного пользовательского интерфейса или предложить сторонним разработчикам создать новые клиентские приложения для Spitter, использующие службу RMI. Разработчики этих клиентских приложений легко смогут реализовать соединение со RMI-службой Spitter, если они также используют фреймворк Spring. А теперь повернемся в другую сторону и посмотрим, как реализовать клиента, пользующегося RMI-службой Spitter.

11.2.2. Внедрение службы RMI

Для поиска службы в реестре RMI клиенты традиционно должны использовать класс `Naming` из RMI API. Например, для получения ссылки на RMI-службу приложения Spitter можно использовать следующий фрагмент:

```
try {
    String serviceUrl = "rmi:/spitter/SpitterService";
    SpitterService spitterService =
        (SpitterService) Naming.lookup(serviceUrl);
    ...
}
```

```
catch (RemoteException e) { ... }
catch (NotBoundException e) { ... }
catch (MalformedURLException e) { ... }
```

Несмотря на то что этот фрагмент позволяет получить ссылку на RMI-службу `spitter`, тем не менее в нем присутствуют две проблемы:

- ❑ в результате типичного поиска службы RMI может быть возбуждено одно из трех контролируемых исключений (`RemoteException`, `NotBoundException` и `MalformedURLException`), которые необходимо перехватить и возбудить заново;
- ❑ ответственность за извлечение службы `spitter` несет сам программный код, пользующийся ею, – это шаблонный код, никак не связанный с функциональностью клиента.

Исключения, возбуждаемые в процессе поиска службы RMI, являются своего рода сигналами о фатальной ошибке в приложении. Исключение `MalformedURLException`, например, указывает на недопустимый адрес службы. Чтобы восстановить работу приложения после этого исключения, необходимо как минимум перенастроить приложение и, возможно, скомпилировать его заново. Никакие ухищрения в блоке `try/catch` не способны исправить эту ошибку, тогда зачем вынуждать программный код перехватывать и обрабатывать его?

Что еще хуже, такой программный код действует в полной противоположности принципам внедрения зависимостей. Поскольку клиент целиком отвечает за поиск службы `Spitter` и эта служба является RMI-службой, нет никакой возможности подставить иную реализацию службы `SpitterService`. В идеале объект `SpitterService` должен был бы внедряться в требуемый компонент, вместо того чтобы заставлять его искать службу самостоятельно. Благодаря поддержке DI любой клиент службы `SpitterService` может вообще не иметь никакого понятия, откуда взялась эта служба.

Компонент `RmiProxyFactoryBean` в Spring – это фабричный компонент, создающий прокси-объект, представляющий службу RMI. С помощью `RmiProxyFactoryBean` легко можно реализовать получение ссылки на RMI-службу `SpitterService`, как показано в следующем фрагменте конфигурационного файла клиентского приложения:

```
<bean id="spitterService"
      class="org.springframework.remoting.rmi.RmiProxyFactoryBean"
      p:serviceUrl="rmi://localhost/SpitterService"
      p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

Адрес URL службы определяется свойством serviceUrl компонента RmiProxyFactoryBean. В данном случае служба имеет имя SpitterService и выполняется на локальном компьютере. Интерфейс, предоставляемый службой, определяется свойством serviceInterface. Порядок взаимодействий между клиентом и прокси-объектом показан на рис. 11.5.

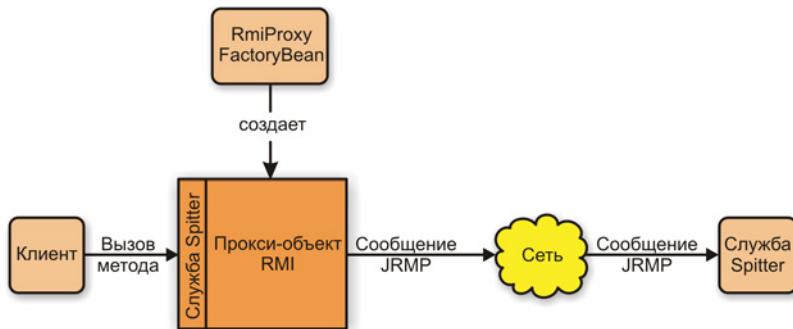


Рис. 11.5. RmiProxyFactoryBean создает прокси-объект, реализующий взаимодействия с удаленными службами RMI от имени клиента. Клиент взаимодействует с прокси-объектом, используя интерфейс службы, как если бы удаленная служба была простым, локальным Java-объектом POJO

Теперь, после объявления службы RMI в виде компонента, управляемого фреймворком Spring, можно внедрить ее в виде зависимости в другой компонент как обычный локальный компонент. Например, предположим, что служба Spitter будет использоваться для извлечения списка сообщений (объектов Spittle), принадлежащих указанному пользователю. Внедрить прокси-объект в клиента можно с помощью аннотации @Autowired:

```
@Autowired
SpitterService spitterService;
```

А затем вызывать методы службы, как если бы она была локальным компонентом:

```
public List<Spittle> getSpittles(String userName) {
    Spitter spitter = spitterService.getSpitter(userName);
    return spitterService.getSpittlesForSpitter(spitter);
}
```

Вся прелесть такого способа обращения к службе RMI состоит в том, что клиенту не требуется даже знать, что он взаимодействует со службой RMI. Он получает объект `SpitterService` посредством внедрения, без необходимости беспокоиться о том, откуда он появился.

Кроме того, прокси-объект автоматически будет перехватывать все исключения `RemoteExceptions` и повторно возбуждать их как неконтролируемые исключения, которые можно попросту игнорировать. Такой подход существенно упрощает замену одной реализации службы другой, которая, в свою очередь, также может быть удаленной службой или фиктивным объектом, используемым для тестирования клиентского программного кода.

Несмотря на то что клиент не знает, что служба `SpitterService`, с которой он взаимодействует, является удаленной службой, вам необходимо со всем вниманием подойти к проектированию интерфейса службы. Обратите внимание, что клиент производит два обращения к службе: один, чтобы получить объект `Spitter` по имени пользователя, и другой, чтобы получить список объектов `Spittle` с сообщениями. Скорость выполнения этих двух вызовов зависит от задержек, связанных с передачей данных по сети, и оказывает влияние на общую производительность клиента. Зная, как будет использоваться служба, имеет смысл перепроектировать интерфейс и объединить эти два вызова в один. Но пока оставим все как есть.

Модель RMI – отличный способ взаимодействия с удаленными службами, но имеет некоторые ограничения. Во-первых, модель RMI значительно сложнее в реализации при наличии сетевых экранов (брандмауэров). Это обусловлено тем, что для взаимодействий RMI используется произвольные порты – что обычно не допускается брандмауэрами. В локальных сетях это не является большой проблемой. Но если предполагается, что взаимодействия будут выполняться через Интернет, вы определенно столкнетесь с проблемами. Даже при том, что RMI поддерживает туннелирование через HTTP (что обычно позволяет преодолевать брандмауэры), настройка туннелирования в RMI – весьма непростая задача.

Еще одна проблема заключается в том, что модель RMI основана на языке Java. То есть и клиент, и служба должны быть реализованы на языке Java. А поскольку в RMI используется механизм сериализации языка Java, версии типов объектов, передаваемых по сети, должны совпадать в обеих взаимодействующих сторонах. Возможно, это и не является проблемой для вашего приложения, но вы долж-



ны помнить об этой особенности, выбирая модель RMI удаленных взаимодействий.

В рамках проекта Caucho Technology (где работают те же люди, создавшие сервер приложений Resin) было разработано решение удаленных взаимодействий, свободное от ограничений, свойственных RMI. В действительности было создано два решения: Hessian и Burlap. Посмотрим, как организовать удаленные взаимодействия с применением Hessian и Burlap в Spring.

11.3. Экспортирование удаленных служб с помощью Hessian и Burlap

Hessian и Burlap – это два решения, созданные в рамках проекта Caucho Technology¹, обеспечивающие реализацию легковесных удаленных служб, действующих по протоколу HTTP. Целью каждого из них является максимальное упрощение API и протоколов веб-служб.

У кого-то может возникнуть вопрос: почему было создано два решения одной и той же проблемы. Hessian и Burlap – это две стороны одной медали, но каждое из этих решений служит немного разным целям. Решение Hessian, подобно RMI, реализует обмен двоичными сообщениями между клиентом и службой. Но, в отличие от других технологий удаленных взаимодействий, использующих двоичные форматы (таких как RMI), двоичный формат Hessian совместим с другими языками программирования, отличными от Java, включая PHP, Python, C++ и C#.

Решение Burlap опирается на обмен данными в формате XML, что автоматически делает его совместимым с любыми языками программирования, способными выполнять парсинг данных в формате XML. А поскольку сообщения передаются в формате XML, они более доступны для человека, чем сообщения в двоичном формате Hessian. В отличие от других технологий удаленных взаимодействий, использующих формат XML (таких как SOAP или XML-RPC), сообщения Burlap имеют весьма простую структуру, и для работы с этим решением не требуется прибегать к внешним определениям на таких языках, как WSDL или IDL.

Кто-то может озадачиться выбором между Hessian и Burlap. Эти два решения в значительной степени идентичны. Единственное от-

¹ <http://www.caucho.com>.

личие – при использовании Hessian сообщения передаются в двоичном формате, а при использовании Burlap – в формате XML. Двоичные сообщения создают меньший трафик. А если для вас важное значение имеет доступность сообщений для восприятия человеком (например, для отладки) или предполагается, что приложение будет взаимодействовать с программами, написанными на других языках программирования, для которых отсутствует реализация решения Hessian, предпочтительнее использовать решение Burlap.

Для демонстрации поддержки решений Hessian и Burlap в Spring вернемся к службе Spitter, реализованной в предыдущем разделе. Но на этот раз посмотрим, как решить ту же проблему с применением Hessian и Burlap в качестве моделей удаленных взаимодействий.

11.3.1. Экспортирование службы с помощью Hessian/Burlap

Как и прежде, предположим, что в качестве службы необходим экспортировать функциональность, реализованную в классе SpitterServiceImpl, но на этот раз с применением технологии Hessian. В этом нет ничего сложного, даже без использования возможностей фреймворка Spring. Достаточно определить класс службы, расширяющий класс com.caucho.hessian.server.HessianServlet, и сделать все методы API службы общедоступными (в Hessian все общедоступные методы считаются методами службы).

Благодаря простоте реализации служб на основе решения Hessian фреймворк Spring делает не так много, чтобы еще больше упростить эту модель. Но при использовании фреймворка Spring служба на основе решения Hessian может использовать все преимущества Spring Framework, отсутствующие в самом решении Hessian. В число этих преимуществ входит использование Spring AOP для организации декларативного управления транзакциями.

Экспортирование службы Hessian

Настройка экспортирования службы Hessian в Spring близко напоминает экспортацию RMI-служб. Чтобы экспортировать компонент службы Spitter в виде RMI-службы, нам потребовалось настроить компонент RmiServiceExporter. Аналогично, чтобы экспортировать службу Spitter в виде службы Hessian, необходимо настроить похожий компонент-экспортатор. На этот раз HessianServiceExporter.

Компонент HessianServiceExporter выполняет те же операции, что и RmiServiceExporter: он экспортирует общедоступные методы POJO как методы службы Hessian. Но, как показано на рис. 11.6, детали операции экспортования отличаются от того, как RmiServiceExporter экспортирует POJO в виде RMI-службы.

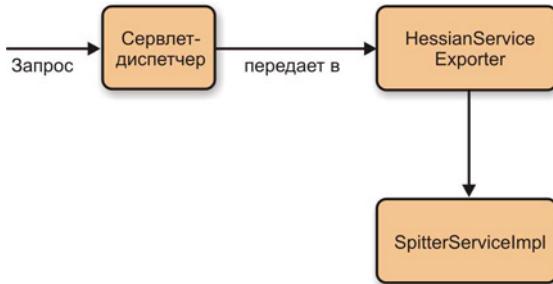


Рис. 11.6. HessianServiceExporter – это контроллер Spring MVC, экспортирующий POJO как службу Hessian, который принимает Hessian-запросы и преобразует их в вызовы методов POJO

Компонент HessianServiceExporter – это контроллер Spring MVC (подробнее об этом чуть ниже), принимающий Hessian-запросы и преобразующий их в вызовы экспортаемых методов POJO.

Следующее объявление компонента HessianServiceExporter экспортирует компонент spitterService как службу Hessian:

```
<bean id="hessianSpitterService"
      class="org.springframework.remoting.caucho.HessianServiceExporter"
      p:service-ref="spitterService"
      p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

Как и в настройках компонента RmiServiceExporter, свойство service служит для внедрения ссылки на компонент, реализующий службу. В данном случае это ссылка на компонент spitterService. Свойство serviceInterface определяет интерфейс, который реализует служба SpitterService.

В отличие от компонента RmiServiceExporter, здесь нет необходимости определять свойство serviceName. В случае со службами RMI свойство serviceName используется для регистрации службы в реестре RMI. Решение Hessian не имеет реестра, и поэтому нет необходимости определять имя службы Hessian.



Настройка контроллера Hessian

Другое важное отличие HessianServiceExporter от RmiServiceExporter обусловлено тем, что решение Hessian основано на использовании протокола HTTP, вследствие чего HessianServiceExporter реализован как контроллер Spring MVC. То есть в процессе экспортирования службы Hessian необходимо выполнить две дополнительные настройки:

- ❑ настроить компонент Spring DispatcherServlet в файле web.xml и развернуть приложение как веб-приложение;
- ❑ настроить обработчик URL в файле конфигурации Spring, чтобы обеспечить обработку URL-адресов службы Hessian в соответствующем компоненте службы Hessian.

Процедура настройки компонента DispatcherServlet и обработчиков URL обсуждалась в главе 8. Поэтому эти настройки должны быть вам знакомы. Во-первых, необходимо определить компонент DispatcherServlet. К счастью, он уже определен в файле web.xml приложения Spitter. Но для поддержки службы Hessian этому компоненту DispatcherServlet требуется сервлет, отображающий URL-адреса *.service:

```
<servlet-mapping>
    <servlet-name>spitter</servlet-name>
    <url-pattern>*.service</url-pattern>
</servlet-mapping>
```

Благодаря этим настройкам любые запросы к указанным URL, оканчивающимся на .service, будут передаваться в DispatcherServlet, который, в свою очередь, передаст их контроллеру, полученному в результате отображения URL. То есть запросы с URL /spitter.service в конечном итоге будут обрабатываться компонентом hessian-SpitterService (который является всего лишь прокси-объектом для SpitterServiceImpl).

Но как обеспечить передачу запросов компоненту hessianSpitterService? Для этого необходимо настроить отображение URL так, чтобы DispatcherServlet отправлял запросы в hessianSpitterService. Все необходимое сделает следующий компонент SimpleUrlHandlerMapping:

```
<bean id="urlMapping" class=
    "org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <value>
```

```
    /spitter.service=hessianSpitterService
  </value>
</property>
</bean>
```

Альтернативой двоичному протоколу Hessian является XML-протокол Burlap. Посмотрим, как организовать экспортацию службы Burlap.

Экспортирование службы Burlap

Компонент BurlapServiceExporter практически идентичен компоненту HessianServiceExporter во всех отношениях, кроме того что вместо двоичного протокола он пользуется протоколом на основе XML. Следующее определение компонента демонстрирует, как экспортовать службу Spitter в виде службы Burlap, используя BurlapServiceExporter:

```
<bean id="burlapSpitterService"
  class="org.springframework.remoting.caucho.BurlapServiceExporter"
  p:service-ref="spitterService"
  p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

Как видно из этого примера, единственное различие между этим компонентом и родственным ему объявлением службы Hessian заключается в идентификаторе компонента и его классе. В остальном настройки службы Burlap ничем не отличаются от настройки службы Hessian, включая настройки обработчика URL и DispatcherServlet.

Теперь обратимся к другой стороне диалога и задействуем службу, экспортованную средствами Hessian (или Burlap).

11.3.2. Доступ к службам Hessian/Burlap

Как было показано в разделе 11.2.2, клиент, пользующийся службой Spitter с применением компонента RmiProxyFactoryBean, не имеет ни малейшего представления, что служба в действительности является RMI-службой. На самом деле он вообще никак не связан с фактической реализацией удаленной службы. Единственное, с чем он имеет дело, – интерфейс SpitterService, а конкретные детали реализации RMI-службы скрыты в настройках компонентов в конфигурационном файле Spring. Вся прелест такой организации состоит в том, что из-за отсутствия в клиенте информации о фактической

реализации службы его легко можно переключить с использования службы RMI на службу Hessian, без необходимости изменять программный код клиента.

Однако для любителей писать программный код на языке Java этот раздел станет полным разочарованием. Это обусловлено тем, что для внедрения в клиента службы Hessian вместо службы RMI достаточно лишь заменить `RmiProxyFactoryBean` на `HessianProxyFactoryBean`. В настройках клиента службу `spitter`, реализованную на основе решения Hessian, можно объявить, как показано ниже:

```
<bean id="spitterService"
    class="org.springframework.remoting.cauch.HessianProxyFactoryBean"
    p:serviceUrl="http://localhost:8080/Spitter/spitter.service"
    p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

Подобно настройкам службы RMI, свойство `serviceInterface` определяет интерфейс, реализованный службой, а свойство `serviceUrl` определяет URL службы. Однако, поскольку решение Hessian основано на протоколе HTTP, в URL указывается протокол HTTP (определяется в настройках отображения адресов URL, выполненных выше). Схема взаимодействий между клиентом и прокси-объектом, создаваемым компонентом `HessianProxyFactoryBean`, изображена на рис. 11.7.

Как оказывается, настройка внедрения службы Burlap в клиента также не содержит ничего интересного. Единственное отличие в том, что вместо `HessianProxyFactoryBean` используется `BurlapProxyFactoryBean`:

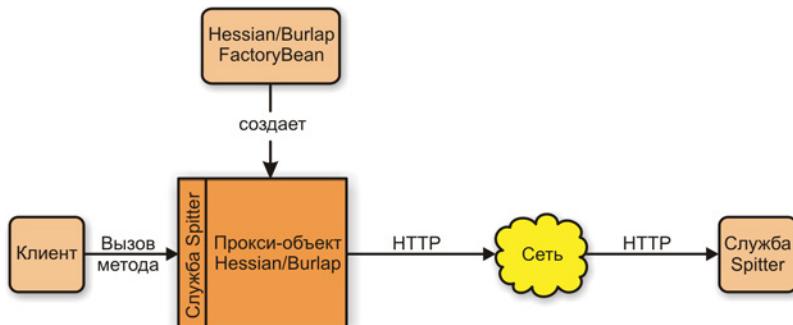


Рис. 11.7. Компоненты `HessianProxyFactoryBean` и `BurlapProxyFactoryBean` создают прокси-объекты, обеспечивающие взаимодействие с удаленной службой по протоколу HTTP (Hessian – в двоичном формате, Burlap – в формате XML)

```
<bean id="spitterService"
      class="org.springframework.remoting.caucho.BurlapProxyFactoryBean"
      p:serviceUrl="http://localhost:8080/Spitter/spitter.service"
      p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

Хотя мы и занялись разбором малоинтересных различий в настройках RMI, Hessian и Burlap, сделано это было совсем не зря. Тем самым была наглядно продемонстрирована простота переключения между различными технологиями удаленных взаимодействий, поддерживаемых фреймворком Spring, без необходимости вникать во все подробности этих моделей. Настроив ссылку на службу RMI, вы практически без труда сможете перенастроить приложение на использование ссылки на службу Hessian или Burlap.

Поскольку решения Hessian и Burlap основаны на использовании протокола HTTP, им не свойственны проблемы с брандмауэрами, характерные для RMI. Однако RMI превосходит оба решения, Hessian и Burlap, когда дело доходит до сериализации объектов, передаваемых в RPC-сообщениях. Решения Hessian и Burlap используют собственный механизм сериализации объектов, тогда как RMI использует механизм, встроенный в Java. При использовании сложных моделей данных, решения Hessian/Burlap могут оказаться недостаточно мощными.

Однако существует решение, заимствованное все самое лучшее из обеих технологий. В следующем разделе мы познакомимся с механизмом Spring HTTP Invoker, обеспечивающим возможность RPC по протоколу HTTP (как Hessian/Burlap) и в то же время использующим механизм сериализации объектов, встроенный в язык Java (как RMI).

11.4. Использование Spring Http Invoker

Разработчики Spring заметили пустующую нишу между службами RMI и службами, основанными на протоколе HTTP, такими как Hessian и Burlap. С одной стороны, в модели RMI используется стандартный для Java механизм сериализации объектов, но она испытывает сложности при наличии брандмауэров в сети. С другой стороны, решения Hessian и Burlap не испытывают проблем с брандмауэрами, но используют свой, нестандартный механизм сериализации объектов.

С учетом всех этих достоинств и недостатков была создана модель Spring HTTP Invoker. HTTP Invoker – это новая модель удален-

ных взаимодействий, реализованная как часть фреймворка Spring Framework и предназначенная для обеспечения взаимодействий по протоколу HTTP (чтобы облегчить жизнь сетевым администраторам), с применением стандартного механизма сериализации языка Java (чтобы облегчить жизнь программистам).

Работа со службами на основе HTTP Invoker напоминает работу со службами Hessian/Burlap. В качестве примера применения механизма HTTP Invoker на практике создадим еще одну службу Spitter, но на этот раз реализуем ее как службу HTTP Invoker.

11.4.1. Экспортирование компонентов в виде служб HTTP Invoker

Чтобы экспортить компонент в виде службы RMI, мы использовали компонент `RmiServiceExporter`. Чтобы экспортить его в виде службы Hessian, мы использовали компонент `HessianServiceExporter`. А чтобы экспортить его в виде службы Burlap, мы использовали компонент `BurlapServiceExporter`. Продолжая этот ряд, совершенно неудивительно, что для экспортования компонента в виде службы Spring HTTP Invoker необходимо использовать компонент `HttpInvokerServiceExporter`.

То есть, чтобы экспортить службу Spitter в виде службы HTTP Invoker, необходимо настроить компонент `HttpInvokerServiceExporter`, как показано ниже:

```
<bean class=
    "org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter"
    p:service-ref="spitterService"
    p:serviceInterface="com.habuma.spitter.service.SpitterService" />
```

У вас не возникло ощущения дежавю? Не так-то просто найти различия между этим объявлением компонента и объявлением в разделе 11.3.2. Единственное отличие – имя класса: `HttpInvokerServiceExporter`. Во всем остальном настройки этого компонента-экспортера ничем не отличаются от настроек других экспортёров службы.

Как показано на рис. 11.8, компонент `HttpInvokerServiceExporter` действует практически так же, как компоненты `HessianServiceExporter` и `BurlapServiceExporter`. Это – контроллер Spring MVC, принимающий запросы от клиента через `DispatcherServlet` и преобразующий эти запросы в вызовы методов POJO, реализующего службу.

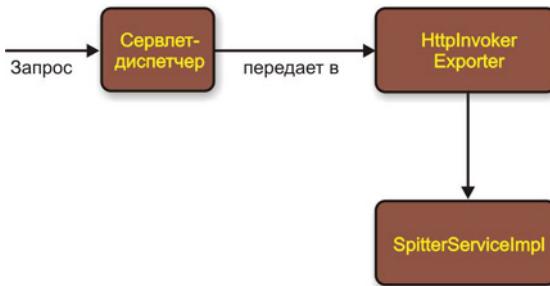


Рис. 11.8. HttpInvokerServiceExporter

действует практически так же, как родственные ему компоненты Hessian и Burlap, принимая запросы и от компонента DispatcherServlet и преобразуя их в вызовы методов компонента, управляемого фреймворком Spring

Поскольку компонент HttpInvokerServiceExporter является контроллером Spring MVC, необходимо настроить обработчик адресов URL, отображающий эти адреса URL на службу, подобно компонентам-экспортерам Hessian и Burlap:

```

<bean id="urlMapping" class=
    "org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <value>
            /spitter.service=httpInvokerSpitterService
        </value>
    </property>
</bean>

```

Кроме того, как и прежде, нужно убедиться, что объявлению компонента DispatcherServlet в файле web.xml сопутствует следующий элемент <servlet-mapping>:

```

<servlet-mapping>
    <servlet-name>spitter</servlet-name>
    <url-pattern>*.service</url-pattern>
</servlet-mapping>

```

При таких настройках служба Spitter будет доступна по тому же адресу URL /spitter.service, который использовался для экспортирования службы с использованием решений Hessian и Burlap.

Выше уже демонстрировалось, как пользоваться удаленными службами RMI, Hessian и Burlap. А теперь переделаем клиента Spitter так, чтобы он использовал только что экспортированную службу HTTP Invoker.

11.4.2. Доступ к службам HTTP Invoker

Рискуя напомнить заезженную пластинку, я все же должен сообщить, что пользование службой HTTP Invoker очень напоминает все, что было показано выше. Как показано на рис. 11.9, компонент `HttpInvokerProxyFactoryBean` играет ту же роль, что и другие прокси-объекты удаленных служб, представленные выше в этой главе.

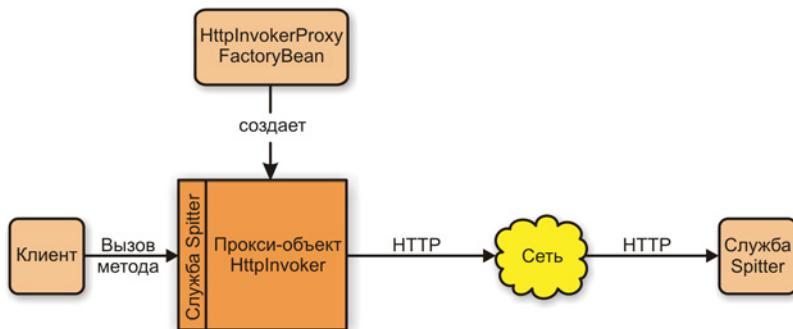


Рис. 11.9. `HttpInvokerProxyFactoryBean` – это фабричный компонент, создающий прокси-объекты, реализующие удаленные взаимодействия по протоколу HTTP

Чтобы внедрить службу HTTP Invoker в контекст клиентского приложения Spring, следует настроить компонент, обернутый прокси-объектом, созданным с помощью `HttpInvokerProxyFactoryBean`, как показано ниже:

```
<bean id="spitterService" class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
    <p:serviceUrl>http://localhost:8080/Spitter/spitter.service</p:serviceUrl>
    <p:serviceInterface>com.habuma.spitter.service.SpitterService</p:serviceInterface>
</bean>
```

Сравните это определение компонента с соответствующими определениями в разделах 11.2.2 и 11.3.2, и вы увидите, что различия минимальны. Свойство `serviceInterface`, как и прежде, определяет



интерфейс, реализованный службой Spitter. А свойство `serviceUrl` определяет адрес удаленной службы. Поскольку модель HTTP Invoker основана на использовании протокола HTTP, подобно решениям Hessian и Burlap, свойство `serviceUrl` может хранить тот же URL, что и версии компонентов на основе решений Hessian и Burlap.

Не любите симметрию?

Механизм Spring HTTP Invoker объединяет в себе лучшие черты решений удаленных взаимодействий – простоту работы с протоколом HTTP и стандартный для Java механизм сериализации объектов. Все это делает HTTP Invoker весьма привлекательной альтернативой RMI или Hessian/Burlap.

Однако модель Http Invoker имеет одно существенное ограничение, о котором необходимо помнить: это решение удаленных взаимодействий предлагается только фреймворком Spring Framework. То есть и клиент, и служба должны быть реализованы как приложения на основе Spring. Это также подразумевает, по крайней мере сейчас, что и клиент, и служба должны быть написаны на языке Java. А поскольку используется стандартный механизм сериализации объектов языка Java, на обеих сторонах должны использоваться одни и те же версии классов (как в RMI).

RMI, Hessian, Burlap и HTTP Invoker – отличные механизмы удаленных взаимодействий. Но когда речь заходит о широкодоступных веб-службах, ни один из них не способен оказать существенную помощь. Далее мы посмотрим, что может предложить фреймворк Spring для поддержки удаленных взаимодействий с веб-службами, основанными на применении протокола SOAP.

11.5. Экспортирование и использование веб-служб

Одной из наиболее популярных трехбуквенных аббревиатур в последние годы стала аббревиатура SOA (Service-Oriented Architecture – сервис-ориентированная архитектура). Для разных людей аббревиатура SOA имеет разное значение. Но в основе SOA лежит идея, заключающаяся в том, что приложения могут и должны разрабатываться с уклоном на использование единого набора основных услуг, вместо того чтобы включать их реализацию в каждое приложение.

Например, в финансовом учреждении может эксплуатироваться несколько различных приложений, требующих доступа к информа-

ции о счетах заемщика. Вместо того чтобы встраивать логику доступа к счетам в каждое приложение (большая часть которой будет просто дублироваться), приложения могли бы использоваться единой службы, извлекающей информацию о счетах.

Java и веб-службы имеют длинную совместную историю, и в Java имеется ряд возможностей, предназначенных для работы с веб-службами. Многие из них так или иначе интегрируются с фреймворком Spring. В этой книге просто невозможно охватить все фреймворки и наборы инструментов для работы с веб-службами, поддерживающие возможность использования совместно с фреймворком Spring. Поэтому далее будет рассмотрена наиболее известная поддержка экспортирования и использования веб-служб SOAP с использованием Java API для XML Web Services, или JAX-WS, входящей в состав самого фреймворка Spring.

А что можно сказать о поддержке JAX-RPC и XFire? В предыдущем издании книги рассказывалось о создании веб-служб с применением XFire (<http://xfire.codehaus.org>) и поддержке JAX-RPC, имеющейся в Spring. В то время эта тема пользовалась большой популярностью, но теперь обе технологии постепенно уходят в прошлое.

Технология JAX-RPC была вытеснена пришедшей ей на смену технологией JAX-WS, ставшей стандартом разработки веб-служб на Java. Фреймворк Spring последовал за общими тенденциями и также отказался от поддержки JAX-RPC в пользу JAX-WS. К счастью, поддержка JAX-WS в Spring очень близко отражает поддержку JAX-RPC. Например, компонент `JaxWsPortProxyFactoryBean` действует практически так же, как старый компонент `JaxRpcPortProxyFactoryBean`.

Технология XFire была моим любимым способом взаимодействий с веб-службами в Spring. Но разработка XFire остановилась на версии 1.2.6. Как полагают многие, версией XFire 2 стал проект Apache CXF (<http://cxf.apache.org>). Поэтому, если вы предпочитаете использовать XFire, подумайте о переходе на проект Apache CXF. Apache CXF – намного более амбициозный проект, чем XFire, однако его изучение выходит далеко за рамки данной книги.

Поскольку одной из моих целей в этом издании было обеспечить максимальную актуальность информации, я решил оставить в стороне технологии JAX-RPC и XFire. Если вам будет интересна любая из этих тем, рекомендую обратиться ко второму изданию книги «Spring in Action». В ней затрагиваются обе темы, а возможности JAX-RPC и XFire с тех пор изменились совсем немного.

В этом разделе мы еще раз вернемся к примеру службы Spitter. Но на этот раз она будет реализована и использована как веб-служба с применением поддержки JAX-WS в Spring. Начнем со знакомства с особенностями создания веб-службы JAX-WS в Spring.



11.5.1. Создание конечных точек JAX-WS с поддержкой Spring

Выше в этой главе мы создавали удаленные службы с использованием компонентов-экспортеров, входящих в состав фреймворка Spring. Эти компоненты волшебным образом преобразуют простые Java-объекты (POJO) в удаленные службы. Мы узнали, как создавать службы RMI с помощью `RmiServiceExporter`, службы Hessian с помощью `HessianServiceExporter`, службы Burlap с помощью `BurlapServiceExporter` и службы HTTP Invoker с помощью `HttpInvokerServiceExporter`. Теперь вы, вероятно, ожидаете, что я покажу, как создавать веб-службы с помощью компонента-экспортера, поддерживающего технологию JAX-WS.

В состав фреймворка Spring входит компонент-экспортер `SimpleJaxWsServiceExporter`, поддерживающий возможность создания веб-служб JAX-WS, и мы вскоре с ним познакомимся. Но прежде вы должны знать, что его применение подходит не во всех ситуациях. Вы увидите, что `SimpleJaxWsServiceExporter` требует, чтобы среда времени выполнения JAX-WS поддерживала публикацию конечных точек для указанного адреса¹. Среда времени выполнения JAX-WS, входящая в состав Sun JDK 1.6, соответствует этому требованию, но другие реализации JAX-WS, включая упомянутую здесь, могут не соответствовать ему.

Если вам придется развертывать среду выполнения JAX-WS, не поддерживающую публикацию службы в указанном адресе, вы должны будете реализовать собственные конечные точки JAX-WS более обычным способом. Это означает, что жизненным циклом конечных точек будет управлять среда выполнения JAX-WS, а не Spring. Но это не означает, что их нельзя будет связать с компонентами в контексте приложения Spring.

Автоматическое связывание конечных точек JAX-WS в Spring

Модель программирования JAX-WS включает в себя использование аннотаций для объявления классов и их методов операциями веб-службы. Класс, отмеченный аннотацией `@WebService`, считается

¹ Точнее, это означает, что провайдер JAX-WS должен поставляться с собственным HTTP-сервером для создания необходимой инфраструктуры публикации службы в требуемом адресе.

конечной точкой веб-службы, а его методы, отмеченные аннотацией @WebMethod, – операциями.

Подобно любым другим объектам в достаточно крупном приложении, конечная точка JAX-WS наверняка будет использовать в своей работе другие объекты. Это означает, что конечные точки JAX-WS с успехом могут пользоваться преимуществами приема внедрения зависимостей. Но поскольку жизненным циклом конечной точки управляет среда выполнения JAX-WS, а не Spring, кажется невозможным обеспечить внедрение компонентов, управляемых фреймворком Spring, в экземпляры конечных точек, управляемых средой выполнения JAX-WS.

Секрет связывания конечных точек JAX-WS заключается в наследовании ими класса SpringBeanAutowiringSupport. Если класс конечной точки будет наследовать класс SpringBeanAutowiringSupport, свойства этого класса можно пометить аннотацией @Autowired и тем самым обеспечить возможность внедрения зависимостей¹. В листинге 11.2 представлен класс SpitterServiceEndpoint, демонстрирующий использование этого приема.

Листинг 11.2. Конечная точка JAX-WS, наследующая класс SpringBeanAutowiringSupport

```
package com.habuma.spitter.remoting.jaxws;
import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.context.support.SpringBeanAutowiringSupport;

import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;
import com.habuma.spitter.service.SpitterService;
```

¹ Хотя в данном случае прием наследования класса SpringBeanAutowiringSupport используется, чтобы обеспечить возможность автоматического связывания для конечных точек JAX-WS, этот же прием с успехом может применяться к любым объектам, управление жизненным циклом которых выполняется за пределами Spring. Единственное требование состоит в том, чтобы контекст приложения Spring и сторонняя среда выполнения находились в пределах одного и того же веб-приложения.



```
@WebService(serviceName="SpitterService")
public class SpitterServiceEndpoint
    extends SpringBeanAutowiringSupport { // Включает поддержку
                                         // автоматического связывания
    @Autowired
    SpitterService spitterService;           // Для внедрения SpitterService

    @WebMethod
    public void addSpittle(Spittle spittle) {
        spitterService.saveSpittle(spittle);
    }

    @WebMethod
    public void deleteSpittle(long spittleId) {
        // Выполнение делегируется службе SpitterService
        spitterService.deleteSpittle(spittleId);
    }

    @WebMethod
    public List<Spittle> getRecentSpittles(int spittleCount) {
        // Выполнение делегируется службе SpitterService
        return spitterService.getRecentSpittles(spittleCount);
    }

    @WebMethod
    public List<Spittle> getSpittlesForSpitter(Spitter spitter) {
        // Выполнение делегируется службе SpitterService
        return spitterService.getSpittlesForSpitter(spitter);
    }
}
```

Здесь свойство `spitterService` отмечено аннотацией `@Autowired`, чтобы показать, что в него автоматически должен внедряться компонент из контекста приложения Spring, которому конечная точка будет делегировать фактическое выполнение операций.

Экспортирование автономных конечных точек JAX-WS

Как уже говорилось, прием наследования класса `SpringBeanAutowiringSupport` удобно использовать для внедрения зависимостей в свойства объектов, управление жизненным циклом которых выполняется за пределами Spring. Но в некоторых ситуациях в качестве конечной точки JAX-WS можно экспортить и компонент, управляемый фреймворком Spring.

Компонент `SimpleJaxWsServiceExporter`, входящий в состав Spring, действует аналогично другим компонентам-экспортерам, обсуждавшимся выше в этой главе, экспортируя компоненты Spring в виде конечных точек служб в среду выполнения JAX-WS. В отличие от других компонентов-экспортеров, `SimpleJaxWsServiceExporter` не требует ссылки на экспортруемый компонент. Вместо этого он публикует все компоненты, отмеченные аннотациями JAX-WS, как службы JAX-WS.

Настройка компонента `SimpleJaxWsServiceExporter` может быть выполнена с помощью элемента `<bean>`, как показано ниже:

```
<bean class=
    "org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter"/>
```

Как видите, компонент `SimpleJaxWsServiceExporter` ничего не требует для своей работы. При запуске он отыскивает в контексте приложения Spring компоненты с аннотацией `@WebService`. Обнаружив такой компонент, он экспортирует его как конечную точку JAX-WS с базовым адресом <http://localhost:8080/>.

В листинге 11.3 демонстрируется один из таких компонентов, обнаруживаемых компонентом `SpitterServiceEndpoint`.

Листинг 11.3. Компонент SimpleJaxWsServiceExporter превращает другие компоненты в конечные точки JAX-WS

```
package com.habuma.spitter.remoting.jaxws;
import java.util.List;

import javax.jws.WebMethod;
import javax.jws.WebService;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.habuma.spitter.domain.Spitter;
import com.habuma.spitter.domain.Spittle;
import com.habuma.spitter.service.SpitterService;

@Component
@WebService(serviceName="SpitterService")
public class SpitterServiceEndpoint {
    @Autowired
```

```
SpitterService spitterService;      // Для внедрения SpitterService

@WebMethod
public void addSpittle(Spittle spittle) {
    // Выполнение делегируется службе SpitterService
    spitterService.saveSpittle(spittle);
}

@WebMethod
public void deleteSpittle(long spittleId) {
    // Выполнение делегируется службе SpitterService
    spitterService.deleteSpittle(spittleId);
}

@WebMethod
public List<Spittle> getRecentSpittles(int spittleCount) {
    // Выполнение делегируется службе SpitterService
    return spitterService.getRecentSpittles(spittleCount);
}

@WebMethod
public List<Spittle> getSpittlesForSpitter(Spitter spitter) {
    // Выполнение делегируется службе SpitterService
    return spitterService.getSpittlesForSpitter(spitter);
}
}
```

Обратите внимание, что эта новая реализация SpitterService-Endpoint больше не наследует класс SpringBeanAutowiringSupport. Как полноценный компонент Spring, она использует механизм автоматического связывания без наследования специального класса поддержки.

Поскольку базовым адресом для SimpleJaxWsServiceEndpoint по умолчанию является адрес <http://localhost:8080/>, а также потому, что SpitterServiceEndpoint отмечен аннотацией @WebService(serviceName="SpitterService"), совокупность этих двух компонентов дает в результате веб-службу, привязанную к URL <http://localhost:8080/SpitterService>. Однако URL службы полностью находится под вашим контролем. То есть при желании можно указать любой другой базовый адрес. Например, следующее определение компонента SimpleJaxWsServiceEndpoint публикует ту же самую конечную точку службы в URL <http://localhost:8888/services/SpitterService>.

```
<bean class=
    "org.springframework.remoting.jaxws.SimpleJaxWsServiceExporter"
    p:baseAddress="http://localhost:8888/services/">
```

Несмотря на простоту настройки компонента SimpleJaxWsServiceEndpoint, следует помнить, что он может использоваться, только если среда выполнения JAX-WS поддерживает публикацию конечных точек в произвольные адреса. Такой поддержкой обладает среда выполнения JAX-WS, входящая в состав Sun 1.6 JDK. Другие реализации JAX-WS, как упоминаемая здесь JAX-WS 2.1, не поддерживают подобного способа публикации конечных точек и потому не могут использоваться совместно с SimpleJaxWsServiceEndpoint.

11.5.2. Проксирование служб JAX-WS на стороне клиента

Экспортирование веб-служб в Spring существенно отличается от экспортирования служб RMI, Hessian, Burlap и Http Invoker. Но, как вскоре будет показано, использование веб-служб с помощью Spring на стороне клиента связано с применением прокси-объектов почти так же, как это делается при использовании других технологий удаленных взаимодействий.

С помощью компонента JaxWsPortProxyFactoryBean можно реализовать внедрение веб-службы Spitter, как если бы это был другой компонент. JaxWsPortProxyFactoryBean – это фабричный компонент, создающий прокси-объекты, способные взаимодействовать с веб-службами по протоколу SOAP. Сам прокси-объект реализует интерфейс службы (рис. 11.10). Таким образом, компонент JaxWsPortProxyFactoryBean делает возможным внедрение и использование удаленной веб-службы, как если бы это был любой другой локальный POJO.

Настройка JaxWsPortProxyFactoryBean для доступа к веб-службе Spitter выполняется следующим образом:

```
<bean id="spitterService"
      class="org.springframework.remoting.jaxws.JaxWsPortProxyFactoryBean"
      p:wsdlDocumentUrl="http://localhost:8080/services/SpitterService?wsdl"
      p:serviceName="spitterService"
      p:portName="spitterServiceHttpPort"
      p:serviceInterface="com.habuma.spitter.service.SpitterService"
      p:namespaceUri="http://spitter.com"/>
```

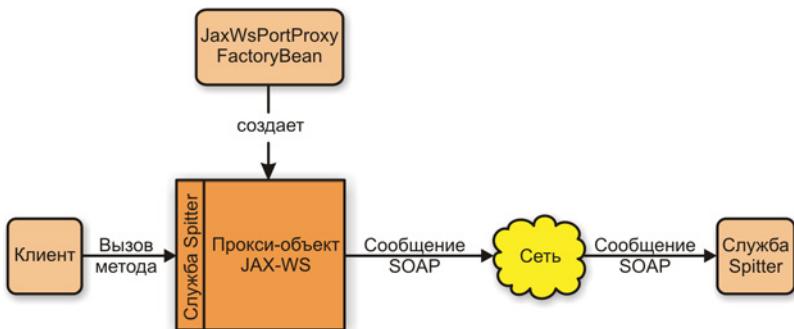


Рис. 11.10. Компонент JaxWsPortProxyFactoryBean создает прокси-объекты, взаимодействующие с удаленными веб-службами. Эти прокси-объекты можно внедрять в другие компоненты, подобно локальным POJO

Как видно из этого примера, для нормальной работы в настройках компонента JaxWsPortProxyFactoryBean необходимо определить значения нескольких свойств. Свойство wsdlDocumentUrl определяет местоположение файла определения удаленной веб-службы. Этот WSDL-файл будет использоваться компонентом JaxWsPortProxyFactoryBean для конструирования прокси-объекта, обеспечивающего доступ к службе. Как указывает свойство serviceInterface, прокси-объект, создаваемый компонентом JaxWsPortProxyFactoryBean, реализует интерфейс SpitterService.

Значения для трех остальных свойств обычно легко можно найти в WSDL-файле определения службы. Для примера предположим, что WSDL-файл службы Spitter содержит следующие строки:

```

<wsdl:definitions targetNamespace="http://spitter.com">
  ...
  <wsdl:service name="spitterService">
    <wsdl:port name="spitterServiceHttpPort"
      binding="tns:spitterServiceHttpBinding">
    ...
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
  
```

Хотя это и не типично, но в одном файле WSDL может быть объявлено несколько служб и/или портов. По этой причине в на-

стройках компонента `JaxWsPortProxyFactoryBean` требуется указывать порты и имена служб в свойствах `portName` и `serviceName` соответственно. Атрибуты `name` в элементах `<wsdl:port>` и `<wsdl:service>` помогут быстро выяснить значения для этих свойств.

Наконец, свойство `namespaceUri` определяет пространство имен службы. Кроме всего прочего, пространство имен помогает компоненту `JaxWsPortProxyFactoryBean` отыскать определение службы в WSDL-файле. Как и в случае с портами и именами служб, вы легко сможете определить значение этого свойства, заглянув в файл WSDL. Обычно пространство имен определяется атрибутом `targetNamespace` элемента `<wsdl:definitions>`.

11.6. В заключение

Реализация взаимодействий с удаленными службами чаще всего является весьма утомительным занятием. Но фреймворк Spring предоставляет поддержку удаленных взаимодействий, которая делает работу с удаленными службами такой же простой, как работа с обычными компонентами JavaBeans.

Для клиентских приложений Spring может предложить фабричные компоненты, создающие прокси-объекты, которые позволяют настраивать использование удаленных служб в приложениях Spring. Независимо от типа используемой службы – RMI, Hessian, Burlap, Spring HTTP Invoker или веб-службы – удаленные службы могут внедряться в приложения как обычные POJO. Более того, фреймворк Spring даже перехватывает все исключения `RemoteExceptions` и повторно возбуждает исключение времени выполнения `RemoteAccessExceptions`, освобождая программиста от необходимости писать шаблонный код обработки исключений, что в большинстве случаев не имеет большого смысла.

Даже при том, что фреймворк Spring скрывает многие детали взаимодействий с удаленными службами, делая их подобными обычным локальным компонентам JavaBeans, не следует забывать о проблемах служб, являющихся следствием их удаленности. Удаленные службы по своей природе обычно менее эффективны, по сравнению с локальными службами. Это обстоятельство следует также учитывать и при разработке удаленных служб, стараясь уменьшить количество необходимых удаленных вызовов, чтобы избежать существенного падения производительности.



В этой главе вы узнали, как с помощью Spring можно экспортировать и использовать службы, основанные на некоторых технологиях удаленных взаимодействий. Хотя эти технологии с успехом можно применять при создании распределенных приложений, тем не менее это лишь верхний слой из всего многообразия технологий, обеспечивающих поддержку сервис-ориентированной архитектуры (Service-Oriented Architecture, SOA).

Здесь также было показано, как экспортировать компоненты в виде веб-служб, действующих по протоколу SOAP. Хотя это и самый простой способ разработки веб-служб, но не самый лучший с архитектурной точки зрения. В следующей главе мы познакомимся с иным подходом к созданию распределенных приложений, позволяющим экспортить части приложения в виде ресурсов RESTful.



Глава 12. Поддержка архитектуры REST в Spring

В этой главе рассматриваются следующие темы:

- ❑ создание контроллеров, играющих роль ресурсов REST;
- ❑ представление ресурсов в форматах XML, JSON и др.;
- ❑ создание REST-клиентов;
- ❑ отправка форм RESTful.

Данные – всему голова.

Разработчикам часто приходится заниматься разработкой сложных приложений, решающих прикладные задачи. Данные – это всего лишь сырье, обрабатываемое приложениями. Но если спросить специалистов в той или иной предметной области, что является более ценным для них, данные или программное обеспечение, они наверняка выберут данные. Данные – это кровь деловой жизни многих компаний. Программное обеспечение часто является взаимозаменяемым. Но данные, накапливаемые в течение нескольких лет, ничем не заменишь¹.

Не кажется ли вам странным, что, несмотря на важность данных, мы часто разрабатываем приложения, не уделяя им должного внимания? Возьмем, например, удаленные службы из предыдущей главы. При их разработке все внимание уделялось функциональности, а не данным и ресурсам.

В последние годы все большую популярность обретает альтернатива традиционным веб-службам на основе протокола SOAP – архитектура *передачи представлений о состоянии* (Representational State Transfer, REST). Чтобы помочь разработчикам приложений на основе фреймворка Spring воспользоваться преимуществами архитектуры REST, ее поддержка была включена в версию Spring 3.0.

¹ Нельзя сказать, что программное обеспечение не имеет никакой ценности. Большинству компаний сложно будет работать без программного обеспечения. Но они погибнут без своих данных.



Самое интересное, что поддержка REST в Spring реализована поверх Spring MVC. То есть мы уже знаем многое из того, что нам потребуется для работы с REST в Spring. В этой главе мы создадим контроллеры, обслуживающие запросы на получение ресурсов RESTful, опираясь на уже имеющиеся знания о фреймворке Spring MVC. А также посмотрим, что может предложить фреймворк Spring для взаимодействия со службами REST со стороны клиента.

Но прежде чем перейти к практической стороне дела, определим, что такое REST.

12.1. Обзор архитектуры REST

Готов поспорить, что вы не впервые слышите или читаете об архитектуре REST. В последние годы разговоры о ней идут непрерывно, и вы наверняка не раз слышали ставшие модными в мире разработки программного обеспечения отзывы об архитектуре REST как о более привлекательной альтернативе веб-службам SOAP.

Конечно, для многих применений, где веб-службы SOAP могут оказаться слишком тяжеловесными, архитектура REST является более простой альтернативой. Но проблема в том, что не все до конца понимают, что же такое REST. В результате эта архитектура оказалась окружена массой ошибочных домыслов. Прежде чем приступить к обсуждению поддержки REST в Spring, необходимо получить общее представление об архитектуре REST и возможностях ее применения.

12.1.1. Основы REST

Многие ошибочно считают, что архитектура REST, являющаяся способом организации «веб-служб с адресами URL», – это еще один механизм вызова удаленных процедур (RPC), подобно SOAP, где вызовы осуществляются посредством простых обращений по протоколу HTTP и без огромных пространств имен SOAP в XML.

В действительности архитектура REST имеет весьма мало общего с RPC. Модель RPC является ориентированной на предоставление услуг и выполнение операций, тогда как архитектура REST ориентирована на предоставление доступа к ресурсам и данным.

Кроме того, хотя в архитектуре REST адреса URL играют важную роль, они – лишь часть общей мозаики. Чтобы понять суть архитектуры REST, попробуем разбить эту аббревиатуру на составные части:

- ❑ *представление* – ресурсы REST могут быть представлены практически в любой форме, включая XML, формат представления объектов JavaScript (JavaScript Object Notation, JSON) или даже HTML, какая лучше подходит для потребителя ресурсов;
- ❑ *состояние* – в архитектуре REST состояние ресурсов представляет больший интерес, чем операции, которые можно выполнить над ресурсами;
- ❑ *передача* – архитектура предусматривает организацию передачи данных ресурса в некотором представлении из одного приложения в другое.

Проще говоря, архитектура REST – это комплекс решений, связанных с передачей информации о состоянии ресурса в некоторой форме от сервера клиенту (или наоборот).

Отталкиваясь от этой точки зрения на архитектуру REST, я постараюсь избегать таких терминов, как «служба REST» или «веб-служба RESTful» и подобных им, которые порождают неправильное представление о преобладающей роли операций. Вместо этого я буду говорить о *ресурсах RESTful*, чтобы подчеркнуть природу REST, ориентированную на ресурсы.

12.1.2. Поддержка REST в Spring

Некоторые ингредиенты, необходимые для экспортирования ресурсов REST, появились в Spring достаточно давно. Но в версии Spring 3.0 появились дополнительные расширения к Spring MVC, обеспечивающие превосходную поддержку REST. Теперь фреймворк Spring поддерживает разработку ресурсов REST, предоставляя следующее:

- ❑ контроллеры, способные обрабатывать все типы HTTP-запросов, включая четыре основных: GET, PUT, DELETE и POST;
- ❑ новая аннотация @PathVariable, позволяющая контроллерам обрабатывать запросы к параметризованным адресам URL (URL, имеющие переменную часть пути);
- ❑ JSP-тег <form:form> из библиотеки тегов JSP, входящей в состав Spring, а также новый компонент HiddenHttpMethodFilter, позволяющий отправлять запросы PUT и DELETE из HTML-форм даже в браузерах, не поддерживающих эти HTTP-запросы;
- ❑ представления и арбитры представлений в Spring, включая новые реализации представлений для отображения моделей данных в форматах XML, JSON, Atom и RSS, позволяют представлять ресурсы в самых разных форматах;



- ❑ с помощью нового арбитра представлений ContentNegotiatingViewResolver можно выбрать формат отображения ресурса, наиболее подходящий для клиента;
- ❑ механизм отображения данных с применением представлений можно полностью обойти с помощью новой аннотации @ResponseBody и различных реализаций интерфейса HttpMethodConverter;
- ❑ аналогично, для преобразования данных HTTP-запроса в Java-объекты, передаваемые методам-обработчикам контроллера, можно воспользоваться новой аннотацией @RequestBody наряду с реализациями интерфейса HttpMethodConverter;
- ❑ класс RestTemplate, упрощающий доступ к ресурсам REST на стороне клиента.

На протяжении всей главы будут исследоваться эти особенности, обеспечивающие поддержку архитектуры REST в Spring, а также демонстрироваться приемы экспортования и использования ресурсов REST. Для начала мы познакомимся с особенностями контроллера Spring MVC, ориентированного на ресурсы.

12.2. Создание контроллеров, ориентированных на ресурсы

Как было показано в главе 8, в Spring MVC используется чрезвычайно гибкая модель создания классов контроллеров. Практически любой метод с практически любой сигнатурой можно объявить обработчиком веб-запросов. Однако из-за такой гибкости фреймворк Spring MVC позволяет создавать контроллеры, которые нелучшим образом подходят для обслуживания ресурсов RESTful. Слишком просто написать контроллер, противоречащий принципам архитектуры REST.

12.2.1. Устройство контроллера, противоречащего архитектуре REST

Чтобы понять, как должны выглядеть контроллеры, поддерживающие архитектуру REST, будет полезно узнать, как выглядят контроллеры, противоречащие этой архитектуре. Примером такого рода контроллеров может служить контроллер DisplaySpittleController, представленный в листинге 12.1.

Листинг 12.1. DisplaySpittleController – контроллер Spring MVC, противоречащий архитектуре REST

```
package com.habuma.spitter.mvc.restless;

import javax.inject.Inject;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

import com.habuma.spitter.service.SpitterService;

@Controller
@RequestMapping("/displaySpittle.htm")           // Отображение URL,
                                                // противоречащее REST
public class DisplaySpittleController {
    private final SpitterService spitterService;

    @Inject
    public DisplaySpittleController(SpitterService spitterService) {
        this.spitterService = spitterService;
    }

    @RequestMapping(method=RequestMethod.GET)
    public String showSpittle(@RequestParam("id") long id, Model model) {
        model.addAttribute(spitterService.getSpittleById(id));
        return "spittles/view";
    }
}
```

Первое, на что следует обратить внимание в листинге 12.1, – имя класса контроллера. Конечно, это всего лишь имя. Но оно точно описывает, что делает этот контроллер. Первое слово *Display* – глагол. Оно наглядно показывает, что этот контроллер ориентирован на выполнение действий, а не на предоставление доступа к ресурсам.

Отметьте аннотацию `@RequestMapping` на уровне класса. Она говорит, что данный контроллер будет обрабатывать запросы к странице `/displaySpittle.htm`. Похоже, что этот контроллер специализируется на представлении сообщений (что подтверждается именем класса). Более того, расширение предполагает, что информация будет отображаться в формате HTML.



Сама реализация контроллера `DisplaySpittleController` не содержит никаких ошибок. Но она противоречит принципам архитектуры REST. Контроллер ориентирован на выполнение конкретной, узкоспециализированной операции: отображение объекта `Spittle` в формате HTML. Даже имя класса явно говорит об этом.

Теперь, когда вы узнали, как выглядит контроллер, противоречащий архитектуре REST, посмотрим, как выглядит контроллер, поддерживающий эту архитектуру. Для начала исследуем, как обрабатываются запросы к адресам URL ресурсов.

12.2.2. Обработка адресов URL в архитектуре *RESTful*

Адреса URL – это первое, о чем думает большинство людей, начиная работать с архитектурой REST. В конце концов, все, что делается в архитектуре REST, делается через URL. Самое забавное, что многие адреса URL обычно делают совсем не то, для чего они предназначены.

URL – это аббревиатура от *Uniform Resource Locator* (унифицированный указатель ресурсов). Согласно этому названию, URL должен служить ссылкой на ресурс. Более того, все адреса URL являются также идентификаторами URI, или *Uniform Resource Identifiers* (унифицированные идентификаторы ресурсов). Если это так, тогда от конкретного URL можно ожидать, что он не только является ссылкой на ресурс, но является еще и его идентификатором.

Тот факт, что URL определяет местоположение ресурса, выглядит вполне естественным. В конце концов, в течение многих лет мы привыкли вводить адреса URL в адресную строку браузера, чтобы отыскать требуемую информацию в Интернете. Но мы не привыкли считать, что URL является также уникальным идентификатором ресурса. Никакие два ресурса не могут размещаться по одному и тому же адресу URL, поэтому URL можно считать средством идентификации ресурса¹.

Многие адреса URL ни на что не указывают и ничего не идентифицируют – они выражают требования. Вместо того чтобы идентифицировать ресурс, они требуют выполнения некоторого действия.

¹ Хоть это и выходит за рамки книги, тем не менее семантика Веб использует идентификационную природу адресов URL для создания связанной сети ссылок на ресурсы.

Например, на рис. 12.1 изображен пример URL, обрабатываемого методом `displaySpittle()` контроллера `DisplaySpittleController`.

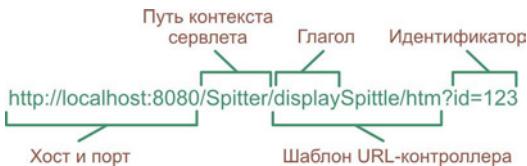


Рис. 12.1. URL, противоречащий архитектуре REST, ориентированной на выполнение операции, а также не ссылающийся и не идентифицирующий ресурс

Как показано на рис. 12.1, этот URL не ссылается и не идентифицирует какой-либо ресурс. Он требует, чтобы сервер отобразил объект `Spittle`. Единственный фрагмент URL, который хоть что-то идентифицирует, – это параметр запроса `id`. Базовая часть URL – глагол, выражающий требование. Все это говорит, что данный URL противоречит архитектуре REST.

Прежде чем писать контроллеры, обрабатывающие адреса URL, полностью соответствующие архитектуре REST, необходимо выяснить, как должны выглядеть такие URL.

Характеристики URL, соответствующих архитектуре REST

В противоположность адресам URL, противоречащим архитектуре REST, адреса URL, соответствующие ей, целиком и полностью признают, что протокол HTTP предназначен для передачи ресурсов. Например, на рис. 12.2 показано, как мог бы выглядеть предыдущий URL после приведения его в согласие с архитектурой REST.

Единственное, что остается неясным в этом URL, – что он делает. Это обусловлено тем, что он ничего *не делает*. Он идентифицирует



Рис. 12.2. URL, указывающий на ресурс и идентифицирующий его



ресурс. В частности, он ссылается на ресурс, представляющий объект `Spittle`. Что будет делаться с этим ресурсом, это уже отдельная тема, зависящая от типа HTTP-запроса (о чем подробнее рассказывается в разделе 12.2.3).

Этот URL не только ссылается на ресурс, но и уникально идентифицирует его. Он в равной степени является адресом URL и идентификатором URI. Для полной идентификации ресурса используется весь URL, а не параметр запроса.

В действительности новый URL вообще не имеет параметров запроса. Несмотря на то что параметры все еще считаются допустимым способом передачи информации на сервер, теперь они являются лишь руководством для сервера, помогающим ему воспроизвести ресурс. Параметры запроса не должны использоваться для идентификации ресурса.

И последнее замечание, касающееся адресов URL в стиле RESTful: обычно они имеют иерархическую организацию. При чтении слева направо более общие понятия сменяются более специализированными. В данном примере URL содержит несколько уровней, любой из которых определяет ресурс.

- ❑ <http://localhost:8080> – определяет доменное имя и порт. Хотя в нашем приложении с этим URL не связано никаких ресурсов, нет объективных причин, почему этого нельзя было бы сделать.
- ❑ <http://localhost:8080/Spitter> – определяет контекст сервлета приложения. Данный URL является более специализированным – он определяет приложение, выполняющееся на сервере.
- ❑ <http://localhost:8080/Spitter/spittles> – определяет ресурс, представляющий список объектов `Spittle` внутри приложения `Spitter`.
- ❑ <http://localhost:8080/Spitter/spittles/123> – наиболее специализированный URL, определяющий конкретный ресурс `Spittle`.

Самая интересная особенность адресов URL в стиле RESTful заключается в том, что пути в них являются параметризованными. В то время как URL, противоречащие архитектуре REST, несут входные данные в виде параметров запроса, адреса URL в стиле RESTful несут эти же данные внутри пути в URL. Для обработки запросов с такими URL необходимо иметь возможность, позволяющую методам-обработчикам контроллеров извлекать входные данные из пути URL.

Встраивание параметров в адреса URL

Для поддержки параметризованных путей в адресах URL в версию Spring 3.0 была добавлена новая аннотация @PathVariable. Чтобы познакомиться с ней в действии, рассмотрим класс SpittleController, новый контроллер Spring MVC, реализующий подход, ориентированный на ресурсы, и обрабатывающий запросы на получение объектов Spittle.

Листинг 12.2. SpittleController – контроллер, поддерживающий архитектуру REST

```
package com.habuma.spitter.mvc;
import javax.inject.Inject;
import javax.validation.Valid;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import com.habuma.spitter.domain.Spittle;
import com.habuma.spitter.service.SpitterService;

@Controller
@RequestMapping("/spittles") // Обрабатывает запросы к URL /spittles
public class SpittleController {
    private SpitterService spitterService;

    @Inject
    public SpittleController(SpitterService spitterService) {
        this.spitterService = spitterService;
    }

    @RequestMapping(value="/{id}", // Используется переменная-заполнитель
                   method=RequestMethod.GET)
    public String getSpittle(@PathVariable("id") long id,
                           Model model) {
        model.addAttribute(spitterService.getSpittleById(id));
        return "spittles/view";
    }
}
```

Как показано в листинге 12.1, класс SpittleController снабжен аннотацией @RequestMapping, указывающей, что данный контроллер бу-



дет обрабатывать запросы на получение ресурсов Spittle – запросов с адресами URL, начинающимися со строки /spittles.

Пока существует только один метод-обработчик, `getSpittle()`. Аннотация `@RequestMapping`, которой отмечен этот метод, в паре с аннотацией `@PathVariable` на уровне класса превращает его в обработчик запросов типа GET с адресами URL вида `/spittles/{id}`.

Возможно, вас заинтересовали странные фигурные скобки в шаблоне URL. Конструкция `{id}` – это переменная-заполнитель, посредством которой изменяющаяся часть URL передается методу. Она соответствует аннотации `@PathVariable`, которой отмечен параметр `id` метода.

Таким образом, если приложение получит запрос GET с URL <http://localhost:8080/Spitter/spittles/123>, метод `getSpittle()` будет вызван со значением 123 в параметре `id`. Затем объект отыщет соответствующий параметру объект `Spittle` и поместит его в модель.

Возможно, вы заметили, что имя `id` трижды используется в сигнатуре метода. Оно используется не только как имя переменной-заполнителя в шаблоне URL и как параметр аннотации `@PathVariable`, оно также используется как имя фактического параметра метода. В данном случае это всего лишь совпадение. Однако если имя параметра метода совпадает с именем переменной-заполнителя в шаблоне URL (я не вижу причин, препятствующих такому совпадению), тогда можно воспользоваться преимуществом наличия принятых соглашений и опустить параметр аннотации `@PathVariable`. Например:

```
@RequestMapping(value="/{id}", method=RequestMethod.GET)
public String getSpittle(@PathVariable long id, Model model) {
    model.addAttribute(spitterService.getSpittleById(id));
    return "spittles/view";
}
```

Когда аннотация `@PathVariable` указывается без параметра, ее параметром становится имя аннотированного параметра метода¹.

Независимо от того, как будет определяться имя переменной-заполнителя, явно или неявно, аннотация `@PathVariable` позволяет пи-

¹ Предполагается, что класс контроллера скомпилирован с сохранением отладочной информации. В противном случае имя параметра метода будет недоступно во время выполнения и аннотация `@PathVariable` не сможет определить имя соответствующей переменной-заполнителя в шаблоне URL.

сать методы контроллеров для обработки запросов, в которых адреса URL идентифицируют ресурс, а не описывают некоторую операцию. С другой стороны, запросы в стиле RESTful – это методы HTTP, которые применяются к адресам URL. Посмотрим, как методы HTTP могут играть роль глаголов, описывающих действия, в запросах REST.

12.2.3. Выполнение операций в стиле REST

Как упоминалось выше, архитектура REST применяется для передачи информации о состоянии ресурса. Поэтому для описания операций над ресурсами достаточно лишь небольшого количества глаголов – глаголов, требующих передачи информации о состоянии ресурса. Наиболее типичными операциями над ресурсами являются создание, чтение, изменение и удаление ресурса.

Глаголы, интересующие нас в данном случае (*послать* (post), *получить* (get), *вставить* (put) и *удалить* (delete)), непосредственно соответствуют четырем методам протокола HTTP, перечисленным в табл. 12.1¹.

Таблица 12.1. Протокол HTTP поддерживает несколько методов управления ресурсами

Метод	Описание	Безопасный	Идемпотентный
GET	Извлекает ресурс с сервера. Ресурс идентифицируется адресом URL в запросе	Да	Да
POST	Посыпает данные на сервер для обработки процессором, ожидающим поступления запросов по адресу URL в запросе	Нет	Нет
PUT	Помещает данные в ресурс на сервере, идентифицируемый адресом URL в запросе	Нет	Да
DELETE	Удаляет ресурс, идентифицируемый адресом URL в запросе	Нет	Да
OPTIONS	Запрашивает дополнительные параметры для взаимодействия с сервером	Да	Да
HEAD	Напоминает метод GET, за исключением того, что в ответ возвращаются только заголовки – содержимое не должно возвращаться в теле ответа	Да	Да
TRACE	В ответ на этот запрос сервер должен вернуть его тело обратно клиенту	Да	Да

¹ Описание протокола HTTP определяет еще четыре метода: TRACE, OPTIONS, HEAD и CONNECT. Но мы сосредоточимся на четырех основных методах.



Каждый из HTTP-методов характеризуется двумя чертами: безопасность и идемпотентность. Метод считается *безопасным*, если он не изменяет состояние ресурса. *Идемпотентный* метод может изменять или не изменять состояние ресурса, но повторные запросы, следующие за первым, не должны оказывать влияния на состояние ресурса. По определению все безопасные методы являются также идемпотентными, но не все идемпотентные методы являются безопасными.

Важно отметить, что хотя фреймворк Spring поддерживает все методы протокола HTTP, это не освобождает разработчика от обязанности следить, чтобы реализация методов контроллеров соответствовала семантике HTTP-методов. Иными словами, метод, обрабатывающий GET-запросы, должен только возвращать ресурс – он не должен изменять или удалять его.

Четыре первых HTTP-метода из перечисленных в табл. 12.1 часто отображаются в *CRUD-операции* (Create/Read/Update/Delete – создать/прочитать/изменить/удалить). В частности, метод GET выполняет операцию чтения, а метод DELETE – операцию удаления. И даже при том, что методы PUT и POST могут использоваться для выполнения других операций, отличных от операций изменения и создания, обычно принято использовать их по прямому назначению.

Выше уже демонстрировался пример обработки GET-запросов. Метод getSpittle() класса SpittleController снабжен аннотацией @RequestMapping, в которой атрибуту method присвоено значение GET. Атрибут method определяет, какой HTTP-метод будет обрабатываться данным методом контроллера.

Изменение ресурса с помощью PUT-запросов

Что касается метода PUT, его семантика полностью противоположна методу GET. В противоположность GET-запросу, требующему передать информацию о состоянии ресурса клиенту, PUT-запрос передает информацию о состоянии ресурса на сервер.

Например, следующий метод putSpittle() предназначен для приема объекта Spittle в составе PUT-запроса:

```
@RequestMapping(value="/{id}", method=RequestMethod.PUT)
@ResponseBody(HttpStatus.NO_CONTENT)
public void putSpittle(@PathVariable("id") long id,
                      @Valid Spittle spittle) {
    spitterService.saveSpittle(spittle);
}
```

Метод `putSpittle()` отмечен аннотацией `@RequestMapping` как любой другой метод-обработчик. Фактически данная аннотация `@RequestMapping` почти ничем не отличается от аннотации для метода `getSpittle()`. Единственное отличие заключается в том, что атрибуту `method` присвоено значение `PUT`.

Поскольку это единственное отличие, следовательно, метод `putSpittle()` будет обрабатывать запросы с адресами URL вида `/spittles/{id}`, аналогично методу `getSpittle()`. Напомню, что URL идентифицирует ресурс, а не операцию над ним. Поэтому URL, идентифицирующий объект `Spittle`, будет тем же самым что для метода `GET`, что для метода `PUT`.

Кроме того, `putSpittle()` метод отмечен аннотацией, не встречавшейся нам до сих пор. Аннотация `@ResponseStatus` определяет код состояния HTTP, который должен быть установлен в ответе, отправляемом клиенту. В данном случае значение `HttpStatus.NO_CONTENT` указывает, что клиенту необходимо вернуть код состояния HTTP `204`. Этот код означает, что запрос был успешно обработан, но тело ответа не содержит никакой дополнительной информации.

Обработка `DELETE`-запросов

Иногда бывает желательно не просто изменить ресурс, а вообще удалить его. В случае с приложением `Spitter`, например, можно дать клиентам возможность удалять сообщения, написанные в спешке, или когда пользователь находился в не совсем адекватном состоянии. Когда надобность в ресурсе отпадает, можно задействовать HTTP-метод `DELETE`.

Для демонстрации обработки `DELETE`-запросов в Spring MVC добавим в класс `SpittleController` новый метод-обработчик, который в ответ на `DELETE`-запросы будет удалять ресурсы `Spittle`:

```
@RequestMapping(value="/{id}", method=RequestMethod.DELETE)
@ResponseStatus(HttpStatus.NO_CONTENT)
public void deleteSpittle(@PathVariable("id") long id) {
    spitterService.deleteSpittle(id);
}
```

И снова аннотация `@RequestMapping` оказалась очень похожей на аналогичные аннотации методов `getSpittle()` и `putSpittle()`. Она отличается только атрибутом `method`, который на этот раз получил значение `DELETE`. Шаблон URL, идентифицирующий ресурс, остался прежним.

Подобно методу `putSpittle()`, метод `deleteSpittle()` также отмечен аннотацией `@ResponseStatus`, чтобы известить клиента об успешной обработке запроса и об отсутствии дополнительной информации в теле ответа.

Создание ресурса с помощью POST-запроса

В каждой компании найдется свой человек, отличающийся свободомыслием и несогласием с остальными. Среди HTTP-методов таким несогласием отличается метод `POST`. Он не подчиняется общепринятым правилам. Он опасен и конечно же не идемпотентен. Этот HTTP-метод нарушает, кажется, все правила, но благодаря этому он выполняет работу, непосильную для других HTTP-методов.

Чтобы увидеть, как действует метод `POST`, посмотрим, как он выполняет работу, которую ему часто поручают, – создание нового ресурса. Метод `createSpittle()`, представленный в листинге 12.3, реализует обработку POST-запросов и создает новые ресурсы `Spittle`.

Листинг 12.3. Создание новых сообщений методом POST

```
@RequestMapping(method=RequestMethod.POST)           // Обрабатывает POST-запросы

@ResponseBody @ResponseStatus(HttpStatus.CREATED)        // Возвращает ответ HTTP 201
public Spittle createSpittle(@Valid Spittle spittle,
                           BindingResult result, HttpServletResponse response)
                           throws BindException {
    if(result.hasErrors()) {
        throw new BindException(result);
    }

    spitterService.saveSpittle(spittle);

    // Указать местоположение ресурса
    response.setHeader("Location", "/spittles/" + spittle.getId());
    return spittle;                                     // Вернуть ресурс
}
```

Первое, на что следует обратить внимание в этом методе, – аннотация `@RequestMapping`, отличающаяся от аналогичных аннотаций, встречавшихся до сих пор. В отличие от них, данная аннотация не имеет атрибута `value`. Это означает, что определение шаблона адресов URL, обрабатываемых методом `createSpittle()`, целиком и полностью возлагается на аннотацию `@RequestMapping` на уровне класса.

Точнее, метод `createSpittle()` будет обрабатывать запросы, соответствующие шаблону URL `/spittles`.

Обычно идентичность ресурса определяется сервером. Поскольку в данном случае создается новый ресурс, нет никакой возможности определить его идентификатор в URL. То есть запросы GET, PUT и DELETE воздействуют непосредственно на ресурс, идентифицируемый адресом URL, а запрос POST вынужден использовать URL, который не является ссылкой на создаваемый ресурс (нельзя определить адрес URL несуществующего ресурса).

И снова метод отмечен аннотацией `@ResponseStatus`, определяющей код состояния в ответе, отправляемом клиенту. На этот раз возвращается код состояния HTTP 201 (создано), свидетельствующий, что ресурс был успешно создан. При возврате кода состояния HTTP 201 вместе с ним необходимо вернуть клиенту и URL нового ресурса. Поэтому в конце метода `createSpittle()` определяется заголовок `Location`, содержащий URL ресурса.

Хотя это и не обязательно, но в теле ответа с кодом HTTP 201 можно вернуть полное представление ресурса. Поэтому, подобно методу `getSpittle()`, обрабатывающему GET-запросы, данный метод завершается, возвращая новый объект `Spittle`. Этот объект будет трансформирован в некоторое представление, которое сможет быть использовано клиентом.

Неясным пока остается сам процесс трансформации. Или как будет выглядеть представление. Рассмотрим букву *R* в аббревиатуре REST: *representation* (представление).

12.3. Представление ресурсов

Представление – важный аспект архитектуры REST, определяющий форму ресурсов при взаимодействиях между клиентом и сервером. Любой ресурс может быть представлен практически в любой форме. Если потребитель ресурса предпочитает формат JSON, ресурс может быть представлен в формате JSON. Если потребитель испытывает слабость к угловым скобкам, тот же самый ресурс может быть представлен в формате XML. Большинство людей, просматривающих ресурсы в веб-браузере, предпочтут получать их в формате HTML (или, может быть, PDF, Excel или каком-то другом удобочитаемом формате). Сам ресурс при этом не изменяется – изменяется только его представление.



Важно помнить, что обычно контроллеры никак не определяют формат представления ресурса. Контроллеры обрабатывают ресурс в терминах Java-объектов. Но как только контроллер завершит свою работу, ресурс тут же будет трансформирован в формат, ожидаемый клиентом.

Фреймворк Spring предоставляет два способа преобразования ресурса из представления на языке Java в представление, которое может быть отправлено клиенту:

- ❑ отображение с помощью представлений на основе договоренностей;
- ❑ преобразование HTTP-сообщений;

В главе 8 мы уже обсуждали арбитры представлений и знаем, как реализовать отображение с помощью представлений (с которыми мы познакомились также в главе 8). Поэтому сначала посмотрим, как обеспечить отображение ресурса в требуемый формат, выбирая представление или арбитра представлений, исходя из договоренностей с клиентом.

12.3.1. Договоренность о представлении ресурса

Как рассказывалось в главе 8, метод-обработчик контроллера обычно возвращает логическое имя представления. Даже если метод не возвращает это имя непосредственно (например, если метод вообще ничего не возвращает), тогда логическое имя представления определяется на основе адреса URL в запросе. Затем сервлет DispatcherServlet передает имя представления арбитру представлений, предлагая ему определить конкретное представление, которое должно использоваться для отображения результатов.

В веб-приложениях, взаимодействующих с человеком, обычно всегда выбирается представление, отображающее результаты в формате HTML. Выбор представления осуществляется по одному параметру – имени представления.

Что касается преобразования имен представлений в фактические представления отображения ресурсов, в операцию выбора добавляется еще один параметр. Представление не только должно соответствовать имени, но и отображать данные в формате, требуемом клиенту. Если клиент требует представить ресурс в формате XML, то представление, возвращающее данные в формате HTML, не годится, даже если оно соответствует указанному имени.

В состав Spring входит класс ContentNegotiatingViewResolver – специализированный арбитр представлений, который при выборе представления учитывает также, в каком формате клиент желает получить ресурс. Подобно любым другим арбитрам представлений, он настраивается в виде компонента в контексте приложения Spring, как показано в листинге 12.4.

Листинг 12.4. ContentNegotiatingViewResolver выбирает наиболее подходящее представление

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
    <property name="mediaTypes">
        <map>
            <entry key="json" value="application/json" />
            <entry key="xml" value="text/xml" />
            <entry key="htm" value="text/html" />
        </map>
    </property>
    <property name="defaultContentType" value="text/html" />
</bean>
```

Чтобы понять принцип действия арбитра ContentNegotiatingViewResolver, необходимо знать, что договоренность о формате ресурса выполняется в два этапа:

1. Определяется тип возвращаемых данных.
2. Отыскивается представление для данного типа.

Рассмотрим подробнее каждый из этих этапов, чтобы лучше понять механизм действия ContentNegotiatingViewResolver. Начнем с определения типа возвращаемых данных.

Определение запрошенного типа возвращаемых данных

Первый этап в процессе определения формата представления ресурса заключается в выяснении требований клиента. На первый взгляд кажется, что в этом нет ничего сложного, поскольку клиент может явно указать требуемый формат в заголовке Accept запроса.

К сожалению, заголовок Accept не является достаточно надежным для этого. Если клиент пользуется веб-браузером, нет никакой гарантии, что клиент желает получить именно то, что браузер отправляет в заголовке Accept. Обычно веб-браузеры указывают форматы, доступные для восприятия человеком (такие как text/html) и



не предоставляют возможности указать другой формат (за исключением случаев использования расширений, предназначенных для разработчиков).

Арбитр `ContentNegotiatingViewResolver` учитывает содержимое заголовка `Accept` и выбирает то или иное представление в соответствии с ним, но только после того, как попытается определить расширение файла в URL. Если в конце URL присутствует расширение файла, будет выполнена попытка отыскать соответствующий ему элемент в свойстве `mediaTypes`. Свойство `mediaTypes` – это отображение, ключами в котором являются расширения имен файлов, а значениями – типы содержимого. При обнаружении совпадения используется соответствующий тип содержимого. При таком подходе расширения имен файлов пользуются более высоким приоритетом по отношению к заголовку `Accept`.

Если имеющееся расширение не соответствует ни одному поддерживаемому типу содержимого, тогда в учет принимается содержимое заголовка `Accept`. Но если запрос не имеет заголовка `Accept`, тогда используется тип содержимого, определяемый свойством `defaultContentType`.

В качестве примера предположим, что арбитр `ContentNegotiatingViewResolver` настроен, как показано в листинге 12.4, и ему предлагается определить тип содержимого по запросу, содержащему расширение файла `.json`. В данном случае этому расширению соответствует элемент с ключом `json` в свойстве `mediaTypes`. Поэтому будет выбран тип содержимого `application/json`.

А теперь допустим, что запрос содержит расширение `.huh`. Этому расширению не соответствует ни один элемент в свойстве `mediaTypes`. В отсутствие совпадения `ContentNegotiatingViewResolver` попытается определить тип содержимого по заголовку `Accept` в запросе. Запросы, отправляемые браузером Firefox, содержат типы `text/html`, `application/xhtml+xml`, `application /xml` и `/*`. Если запрос не содержит заголовка `Accept`, тогда будет выбран тип `text/html`, согласно значению свойства `defaultContentType`.

Изменение порядка определения типа содержимого

Порядок определения типа содержимого, описанный выше, соответствует стратегии по умолчанию. Однако существуют дополнительные возможности, позволяющие влиять на этот порядок:

- ❑ присвоив свойству favorPathExtension значение false, можно заставить ContentNegotiatingViewResolver игнорировать расширение файла в URL;
- ❑ добавив фреймворк Java Activation Framework (JAF) в библиотеку классов (classpath), можно заставить ContentNegotiatingViewResolver обращаться к JAF за помощью в определении типа содержимого по расширению имени файла, если в свойстве mediaTypes не будет найден соответствующий элемент;
- ❑ если присвоить свойству favorParameter значение true, тип содержимого будет определяться путем сопоставления параметра format в запросе (если присутствует) с элементами в свойстве mediaTypes (кроме того, имя параметра можно изменить, определив свойство parameterName);
- ❑ присвоив свойству ignoreAcceptHeader значение true, можно исключить из рассмотрения заголовок Accept.

Например, допустим, что свойству favorParameter было присвоено значение true:

```
<property name="favorParameter" value="true" />
```

Теперь запросы, в которых URL не содержит расширения имени файла, будут соответствовать типу содержимого application/json, если URL включает параметр запроса format со значением json.

После того как ContentNegotiatingViewResolver определит тип содержимого, можно приступать к поиску представления, которое сможет отобразить данные в содержимое этого типа.

Поиск представления

В отличие от других арбитров представлений, ContentNegotiatingViewResolver не определяет представление непосредственно, а предоставляет другим арбитрам представлений возможность выбрать наиболее подходящее представление, соответствующее требованиям клиента. Если не оговаривается иное, он будет использовать любые арбитры представлений, имеющиеся в контексте приложения. Но имеется возможность явно перечислить арбитры представлений, которые следует задействовать, перечислив их в свойстве viewResolvers.

ContentNegotiatingViewResolver опросит все арбитры представлений, предложив им определить представление по логическому имени, и поместит полученные представления в список кандидатов. Кроме



того, если определено свойство `defaultView`, представление, указанное в нем, также будет добавлено в конец списка.

После составления списка кандидатов `ContentNegotiatingViewResolver` выполнит обход по всем запрашиваемым типам содержимого, пытаясь отыскать соответствующее представление из числа кандидатов. Поиск производится до первого совпадения.

Если представление определить не удалось, `ContentNegotiatingViewResolver` вернет пустую ссылку (`null`). Или, если свойство `useNotAcceptableStatusCode` имеет значение `true`, будет возвращено представление с кодом состояния HTTP 406 (Not Acceptable).

Прием определения договоренности о формате представления ресурса прекрасно вписывается в схему, которая использовалась при разработке веб-интерфейса нашего приложения в главе 8. Он позволяет создавать новые представления в добавок к уже имеющимся HTML-представлениям.

При определении ресурсов в стиле RESTful, потребляемых программными клиентами, возможно, имеет смысл создать контроллер, который будет осведомлен, что данные потребляются как ресурс другим приложением. В этом случае можно задействовать инструменты преобразования HTTP-сообщений и аннотацию `@ResponseBody`.

12.3.2. Преобразование HTTP-сообщений

Как было показано в главе 8 и в предыдущем разделе, типичный метод контроллера Spring MVC завершается записью некоторых данных в модель и определением логического имени представления для отображения этих данных. Несмотря на большое разнообразие способов заполнения модели данными и идентификации представлений, все методы-обработчики контроллеров, встречавшиеся нам до сих пор, следовали этому основному шаблону.

Но когда задача контроллера состоит в том, чтобы воспроизвести ресурс в некотором формате, существует более прямой путь к цели, минуя модели и представления. При таком подходе к реализации метода-обработчика возвращаемый им объект автоматически преобразуется в формат, запрошенный клиентом.

Использование этого нового приема начинается с применения аннотации `@ResponseBody` к методу-обработчику контроллера.

Возврат ресурса в теле ответа

Обычно, когда метод-обработчик возвращает Java-объект (любого типа, отличного от `String`), этот объект помещается в модель для

последующего отображения в представлении. Но если метод отметить аннотацией @ResponseBody, тогда возвращаемый им объект будет передан механизму преобразования HTTP-сообщений и превращен в формат, требуемый клиентом.

Например, рассмотрим следующий метод getSpitter() класса SpitterController:

```
@RequestMapping(value = "/{username}", method = RequestMethod.GET,
    headers = {"Accept=text/xml, application/json"})
public @ResponseBody
Spitter getSpitter(@PathVariable String username) {
    return spitterService.getSpitter(username);
}
```

Аннотация @ResponseBody сообщает фреймворку Spring, что возвращаемый объект следует отправить клиенту как ресурс, преобразовав его в некоторый формат, доступный для клиента. Точнее, ресурс должен быть преобразован в формат в соответствии с содержимым заголовка Accept. Если в запросе отсутствует заголовок Accept, тогда предполагается, что клиент способен принимать ресурсы в любом формате.

Что касается заголовка Accept, обратите внимание на аннотацию @RequestMapping перед методом getSpitter(). Атрибут headers указывает, что этот метод будет обрабатывать только запросы, в которых заголовок Accept включает text/xml или application/json. Любые другие запросы, даже если это будут GET-запросы, в которых URL соответствует указанному шаблону, не будут обрабатываться данным методом. Они либо будут обработаны другим методом (если имеется соответствующий метод), либо клиенту будет отправлен ответ с кодом состояния HTTP 406 (Not Acceptable).

Преобразование произвольных Java-объектов, возвращаемых методами обработчиками, в представление, доступное для клиента, выполняется одним из преобразователей HTTP-сообщений, входящих в состав Spring и перечисленных в табл. 12.2.

Например, допустим, что в заголовке Accept запроса клиент сообщил, что он способен принимать данные в формате application/json. Допустим также, что в библиотеке классов приложения присутствует библиотека Jackson JSON. В этом случае объект, возвращаемый методом-обработчиком, можно передать преобразователю MappingJacksonHttpMessageConverter для преобразования его в формат JSON перед передачей клиенту. С другой стороны, если заголовок в запросе ука-

Таблица 12.2. Фреймворк Spring предоставляет несколько преобразователей HTTP-сообщений, способных преобразовывать данные различных Java-типов в представления ресурсов и обратно

Преобразователь	Назначение
AtomFeedHttpMessageConverter	Преобразует объекты класса Feed из библиотеки Rome ¹ в ленту Atom и обратно (тип application/atom+xml). Регистрируется, если библиотека Rome присутствует в библиотеке классов
BufferedImageHttpMessageConverter	Преобразует BufferedImages в двоичные изображения и обратно
ByteArrayHttpMessageConverter	Читает/записывает массивы байт. Читает содержимое всех типов (*) и записывает как содержимое типа application/octet-stream. Регистрируется по умолчанию
FormHttpMessageConverter	Преобразует содержимое типа application/x-www-form-urlencoded в объект MultiValueMap<String, String>, а также преобразует объекты типа MultiValueMap<String, String> в содержимое application/x-www-form-urlencoded и MultiValueMap<String, Object> в multipart/form-data
Jaxb2RootElementHttpMessageConverter	Преобразует данные из формата XML (text/xml или application/xml) в JAXB2-объекты и обратно. Регистрируется, если библиотека JAXB v2 присутствует в библиотеке классов
MappingJacksonHttpMessageConverter	Преобразует данные из формата JSON в типизированные объекты или нетипизированные объекты HashMap и обратно. Регистрируется, если библиотека Jackson JSON присутствует в библиотеке классов
MarshallingHttpMessageConverter	Преобразует данные из формата XML и обратно с использованием внедряемых компонентов преобразования. В число поддерживаемых входят компоненты из библиотек Castor, JAXB2, JIBX, XMLBeans и XStream
ResourceHttpMessageConverter	Преобразует объекты типа Resource. Регистрируется по умолчанию

¹ <https://rome.dev.java.net>.

Таблица 12.2 (окончание)

RssChannelHttpMessageConverter	Преобразует объекты класса Channel из библиотеки Rome в ленту RSS и обратно. Регистрируется, если библиотека Rome присутствует в библиотеке классов
SourceHttpMessageConverter	Преобразует данные из формата XML в объекты javax.xml.transform.Source и обратно. Регистрируется по умолчанию
StringHttpMessageConverter	Преобразует содержимое всех типов (/*) в объекты String. Преобразует объекты Strings в содержимое типа text/plain. Регистрируется по умолчанию
XmlAwareFormHttpMessageConverter	Расширение преобразователя FormHttpMessageConverter, добавляет поддержку преобразования фрагментов в формате XML с использованием SourceHttpMessageConverter. Регистрируется по умолчанию

зывает, что клиент предпочитает формат text/xml, преобразование данных в формат XML можно поручить преобразователю Jaxb2Root ElementHttpMessageConverter.

Обратите внимание, что преобразователи HTTP-сообщений из табл. 12.2 (кроме трех) регистрируются по умолчанию, поэтому их не требуется настраивать отдельно. Однако для их поддержки может понадобиться добавить дополнительные библиотеки в библиотеку классов приложения (classpath). Например, для преобразования данных в формат JSON и обратно с помощью преобразователя MappingJacksonHttpMessageConverter необходимо будет добавить библиотеку Jackson JSON Processor¹.

Прием ресурса в теле запроса

Другому участнику диалога в стиле RESTful, клиенту, может потребоваться отправить на сервер объект в формате JSON, XML или каком-то другом. Было бы неудобно получать эти объекты в исходном виде в методе контроллера и пытаться преобразовать их вручную. К счастью, аннотация @RequestBody позволяет выполнить все необходимые преобразования объектов, отправленных клиентом, как это делает аннотация @ResponseBody с объектами, возвращаемыми клиентам.

¹ <http://jackson.codehaus.org>.



Допустим, что клиент отправил запрос PUT с данными для объекта Spitter в формате JSON. Чтобы принять это сообщение как объект Spitter, достаточно всего лишь отметить соответствующий параметр типа Spitter метода-обработчика аннотацией @RequestBody:

```
@RequestMapping(value = "/{username}", method = RequestMethod.PUT,
                 headers = "Content-Type=application/json")
@ResponseBody(HttpStatus.NO_CONTENT)
public void updateSpitter(@PathVariable String username,
                           @RequestBody Spitter spitter) {
    spitterService.saveSpitter(spitter);
}
```

Получив запрос, фреймворк Spring MVC определит, что этот запрос должен обрабатывать метод updateSpitter(). Но полученное сообщение имеет формат JSON, а метод ожидает получить объект Spitter. В этом случае для преобразования сообщения из формата JSON в объект Spitter можно было бы задействовать преобразователь MappingJacksonHttpMessageConverter. Чтобы этот выбор состоялся, должны быть выполнены следующие условия:

- заголовок Content-Type запроса должен иметь значение application/json;
- библиотека Jackson JSON должна находиться в библиотеке классов приложения.

Возможно, вы обратили внимание, что метод updateSpitter() также отмечен аннотацией @ResponseStatus. После обработки запроса PUT не требуется возвращать клиенту какие-либо данные, а аннотирование метода updateSpitter() таким способом обеспечивает отправку клиенту кода состояния HTTP 204 (No Content).

К настоящему моменту у нас имеются несколько контроллеров Spring MVC с методами-обработчиками для обработки запросов на получение ресурсов. Нам предстоит обсудить еще ряд тем, касающихся определения RESTful API с помощью Spring MVC, и мы вернемся к этому обсуждению в разделе 12.5. Но перед этим немного отвлечемся и посмотрим, как можно использовать класс RestTemplate для создания клиентов, потребляющих ресурсы.

12.4. Клиенты REST

Обычно в нашем представлении веб-приложения ассоциируются с пользовательским интерфейсом в веб-браузере. Но к веб-прило-

жениям, реализующим работу с ресурсами в стиле RESTful, это не относится. Сам факт передачи данных через Всемирную паутину еще не означает, что эти данные обязательно должны отображаться в окне веб-браузера. Более того, можно даже написать веб-приложение, взаимодействующее с другим веб-приложением посредством RESTful API.

Разработка программ, взаимодействующих с ресурсами в стиле RESTful, может оказаться весьма утомительным занятием, требующим писать массу шаблонного кода. Например, допустим, что потребовалось написать некоторый клиентский программный код, который пользовался бы прикладным интерфейсом извлечения сообщений определенного пользователя, разработанным выше в этой главе. В листинге 12.5 представлено одно из возможных решений этой задачи.

Листинг 12.5. При создании клиента в архитектуре REST придется написать массу шаблонного кода и предусмотреть обработку исключений

```
public Spittle[] retrieveSpittlesForSpitter(String username) {  
    try {  
        HttpClient httpClient = new DefaultHttpClient();  
  
        String spittleUrl = "http://localhost:8080/Spitter/spitters/" +  
                            username + "/spittles"; // Подготовить URL  
  
        HttpGet getRequest = new HttpGet(spittleUrl); // Создать запрос  
  
        getRequest.setHeader(  
            new BasicHeader("Accept", "application/json"));  
  
        HttpResponse response = httpClient.execute(getRequest); // Выполнить  
  
        HttpEntity entity = response.getEntity(); // Извлечь результаты  
        ObjectMapper mapper = new ObjectMapper();  
        return mapper.readValue(entity.getContent(), Spittle[].class);  
    } catch (IOException e) {  
        throw new SpitterClientException("Unable to retrieve Spittles", e);  
    }  
}
```

Как видите, чтобы получить ресурс REST, необходимо немало потрудиться. И это при том, что я немного сжульничал, воспользовавшись



вавшись библиотекой Jakarta Commons HTTP Client¹, чтобы создать запрос, и библиотекой Jackson JSON Processor² для преобразования ответа.

Взглянув поближе на метод `retrieveSpittlesForSpitter()`, можно заметить, что здесь не так много программного кода связано с реализацией конкретной функциональностью. Если теперь написать другой метод, извлекающий другой ресурс REST, обнаружится, что эти два метода имеют совсем немного отличий.

Более того, метод выполняет множество операций, каждая из которых может возбудить исключение `IOException`. Поскольку `IOException` является контролируемым исключением, я вынужден предусмотреть либо его обработку, либо повторное его возбуждение. В данном случае я предпочел перехватить исключение и взамен возбудить неконтролируемое исключение `SpitterClientException`.

С таким количеством шаблонного кода, связанного с извлечением ресурса, было бы неплохо иметь возможность инкапсулировать его и обеспечить различные варианты выполнения с помощью параметров. Именно эту цель преследует класс `RestTemplate`, входящий в состав фреймворка Spring. Подобно тому, как `JdbcTemplate` прячет неудобства работы с данными при помощи JDBC, `RestTemplate` освобождает программиста от реализации рутинных операций при работе с ресурсами RESTful.

Чуть ниже будет показано, как можно переписать метод `retrieveSpittlesForSpitter()` и с помощью `RestTemplate` существенно уменьшить объем шаблонного кода. Но сначала познакомимся со всеми REST-операциями, которые может предложить класс `RestTemplate`.

12.4.1. Операции класса `RestTemplate`

В табл. 12.1 был представлен список из семи методов HTTP, применяемых для взаимодействия с ресурсами RESTful. Эти методы играют роль глаголов в диалоге в стиле RESTful.

Класс `RestTemplate` определяет 33 метода и использует все методы HTTP для взаимодействия с ресурсами REST. К сожалению, в этой главе не так много места, чтобы можно было подробно рассмотреть все 33 метода. Но, как оказывается, в действительности класс поддерживает всего 11 уникальных операций, каждая из которых имеет три реализации в виде перегруженных методов. Список из 11 уни-

¹ <http://hc.apache.org/httpcomponents-client/index.html>.

² <http://jackson.codehaus.org/>.

кальных операций, поддерживаемых классом RestTemplate, представлен в табл. 12.3.

Таблица 12.3. Класс RestTemplate определяет 11 уникальных операций, каждая из которых поддерживается тремя перегруженными методами общим числом 33

Метод	Описание
delete()	Выполняет запрос HTTP DELETE к ресурсу с указанным URL
exchange()	Выполняет HTTP-запрос требуемого типа к ресурсу с указанным URL и возвращает экземпляр ResponseEntity, содержащий объект, отображающий тело ответа
execute()	Выполняет HTTP-запрос требуемого типа к ресурсу с указанным URL, возвращает объект, отображающий тело ответа
getForEntity()	Выполняет запрос HTTP GET и возвращает экземпляр ResponseEntity, содержащий объект, отображающий тело ответа
getForObject()	Выполняет запрос HTTP GET и возвращает объект, отображающий тело ответа
headForHeaders()	Выполняет запрос HTTP HEAD к ресурсу с указанным URL и возвращает заголовки ответа
optionsForAllow()	Выполняет запрос HTTP OPTIONS и возвращает заголовок Allow для указанного URL
postForEntity()	Выполняет запрос HTTP POST и возвращает экземпляр ResponseEntity, содержащий объект, отображающий тело ответа
postForLocation()	Выполняет запрос HTTP POST и возвращает URL нового ресурса
postForObject()	Выполняет запрос HTTP POST и возвращает объект, отображающий тело ответа
put()	Выполняет запрос HTTP PUT, отправляя измененный ресурс с указанным URL

Класс RestTemplate использует все методы HTTP, за исключением TRACE. Кроме того, методы execute() и exchange() предлагают возможность выполнения любых HTTP-запросов.

Каждая операция из представленных в табл. 12.3 реализована в форме трех перегруженных методов:

- один принимает URL в виде java.net.URI без поддержки параметризованных адресов URL;
- один принимает URL в виде строки с параметрами URL в виде экземпляра Map;

- один принимает URL в виде строки, с параметрами URL в виде списка аргументов переменной длины.

Познакомившись с 11 операциями, предоставляемыми классом RestTemplate, и особенностями их использования, вы без труда сможете создавать собственные клиентские приложения для работы с ресурсами REST. Познакомимся поближе с операциями, предлагаемыми классом RestTemplate, которые используют четыре основных метода HTTP: GET, PUT, DELETE и POST. И начнем знакомство с операций getForObject() и getForEntity(), использующих метод GET.

12.4.2. Чтение ресурсов

Вы могли заметить, что в табл. 12.3 перечислены два метода, выполняющих GET-запросы: getForObject() и getForEntity(). Как отмечалось выше, каждый из этих методов имеет три перегруженные версии. Ниже приводятся сигнатуры трех версий метода getForObject():

```
<T> T getForObject(URI url, Class<T> responseType)
                  throws RestClientException;

<T> T getForObject(String url, Class<T> responseType,
                  Object... uriVariables) throws RestClientException;

<T> T getForObject(String url, Class<T> responseType,
                  Map<String, ?> uriVariables) throws RestClientException;
```

И сигнатуры трех версий метода getForEntity():

```
<T> ResponseEntity<T> getForEntity(URI url, Class<T> responseType)
                  throws RestClientException;

<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
                  Object... uriVariables) throws RestClientException;

<T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
                  Map<String, ?> uriVariables) throws RestClientException;
```

За исключением типа возвращаемого значения, методы getForObject() являются зеркальным отражением методов getForEntity(). И в действительности они действуют точно так же. Оба метода выполняют GET-запрос, чтобы извлечь ресурс с указанным URL. И оба отображают этот ресурс в объект некоторого типа, определяемого

параметром `responseType`. Единственное отличие состоит в том, что метод `getForObject()` просто возвращает объект требуемого типа, а метод `getForEntity()` возвращает этот объект наряду с дополнительной информацией об ответе.

Рассмотрим сначала более простой метод `getForObject()`. Затем узнаем, как получить дополнительную информацию из ответа на GET-запрос с помощью метода `getForEntity()`.

Извлечение ресурсов

Метод `getForObject()` является сугубо прагматичным способом извлечения ресурсов. С его помощью программа запрашивает ресурс и получает его в форме объекта указанного типа. В качестве простого примера использования метода `getForObject()` рассмотрим еще одну версию метода `retrieveSpittlesForSpitter()`:

```
public Spittle[] retrieveSpittlesForSpitter(String username) {  
    return new RestTemplate().getForObject(  
        "http://localhost:8080/Spitter/spitters/{spitter}/spittles",  
        Spittle[].class, username);  
}
```

Сравните его с листингом 12.5, где реализация `retrieveSpittlesForSpitter()` содержит более десятка строк кода. Применение класса `RestTemplate` позволило сократить метод до нескольких строк (и строк было бы еще меньше, если бы их не нужно было переносить, чтобы уместить по ширине книжной страницы).

Новая версия метода `retrieveSpittlesForSpitter()` начинается с создания экземпляра `RestTemplate` (при желании можно было бы использовать внедренный экземпляр), а затем вызывается метод `getForObject()`, чтобы извлечь список сообщений. То есть запрашивается массив объектов `Spittle`. После получения массива он возвращается вызывающей программе.

Обратите внимание, что для конструирования URL в этой новой версии метода `retrieveSpittlesForSpitter()` не используется операция конкатенации строк. Вместо этого используется тот факт, что `RestTemplate` принимает параметризованные адреса URL. Переменная-заполнитель `{spitter}` в URL будет замещена значением параметра `username` метода. Последний аргумент метода `getForObject()` – это список аргументов переменной длины, где каждый аргумент замещает переменные-заполнители в URL в порядке их следования.

При желании можно было бы поместить параметр `username` в отображение Map с ключом `spitter` и передать это отображение методу `getForObject()` в последнем параметре:

```
public Spittle[] retrieveSpittlesForSpitter(String username) {  
    Map<String, String> urlVariables = new HashMap<String, String>();  
    urlVariables.put("spitter", username);  
    return new RestTemplate().getForObject(  
        "http://localhost:8080/Spitter/spitters/{spitter}/spittles",  
        Spittle[].class, urlVariables);  
}
```

Здесь отсутствует явное преобразование данных в формате JSON в объект. Преобразование тела ответа в требуемый объект выполняется методом `getForObject()` автоматически. Это преобразование выполняется с привлечением все тех же преобразователей HTTP-сообщений, перечисленных в табл. 12.2, которые Spring MVC использует в методах-обработчиках, отмеченных аннотацией `@ResponseBody`.

В этом методе также отсутствует обработка исключений. И вовсе не потому, что метод `getForObject()` не может возбуждать исключения, а потому, что любые исключения, возбуждаемые им, являются неконтролируемыми. Если в методе `getForObject()` что-то пойдет не так, он возбудит неконтролируемое исключение `RestClientException`. При желании его можно перехватить, но компилятор не вынуждает делать это.

Извлечение метаданных из ответа

В качестве альтернативы методу `getForObject()` класс `RestTemplate` предлагает метод `getForEntity()`. Этот метод действует практически так же, как метод `getForObject()`. Но если метод `getForObject()` возвращает только ресурс (преобразованный в Java-объект с помощью преобразователя HTTP-сообщений), то метод `getForEntity()` возвращает тот же самый объект, помещенный внутрь объекта `ResponseEntity`. Объект `ResponseEntity` несет в себе также дополнительную информацию об ответе, такую как код состояния HTTP и заголовки ответа.

Объект `ResponseEntity` можно использовать, чтобы извлечь значение какого-либо заголовка ответа. Например, представьте, что помимо ресурса необходимо также определить, когда ресурс изменился в последний раз. Допустим, что сервер предоставляет такую инфор-

мацию в заголовке `Last-Modified`. Тогда, чтобы получить требуемую информацию, можно воспользоваться методом `getHeaders()`, как показано ниже:

```
Date lastModified = new Date(response.getHeaders().getLastModified());
```

Метод `getHeaders()` возвращает объект `HttpHeaders`, имеющий несколько вспомогательных методов для извлечения заголовков, включая метод `getLastModified()`, возвращающий количество миллисекунд, прошедших с 1 января 1970 года.

Кроме метода `getLastModified()`, класс `HttpHeaders` содержит следующие методы:

```
public List<MediaType> getAccept() { ... }
public List<Charset> getAcceptCharset() { ... }
public Set<HttpMethod> getAllow() { ... }
public String getCacheControl() { ... }
public long getContentLength() { ... }
public MediaType getContentType() { ... }
public long getDate() { ... }
public String getETag() { ... }
public long getExpires() { ... }
public long getIfNotModifiedSince() { ... }
public List<String> getIfNoneMatch() { ... }
public long getLastModified() { ... }
public URI getLocation() { ... }
public String getPragma() { ... }
```

Более универсальный способ доступа к HTTP-заголовкам обеспечивают методы `get()` и `getFirst()`. Оба принимают строковый аргумент, определяющий имя заголовка. Метод `get()` возвращает список строковых значений – по одному для каждого заголовка. Метод `getFirst()` возвращает значение только первого заголовка.

Чтобы получить код состояния HTTP, можно воспользоваться методом `getStatusCode()`. Например, взгляните на реализацию метода `retrieveSpittlesForSpitter()` в листинге 12.6.

Листинг 12.6. Объект `ResponseEntity` включает код состояния HTTP

```
public Spittle[] retrieveSpittlesForSpitter(String username) {
    ResponseEntity<Spittle[]> response = new RestTemplate().getForEntity(
        "http://localhost:8080/Spitter/spitters/{spitter}/spittles",
        Spittle[].class, username);
```

```
if(response.getStatusCode() == HttpStatus.NOT_MODIFIED) {  
    throw new NotModifiedException();  
}  
return response.getBody();  
}
```

Если сервер вернет код состояния HTTP 304, это будет свидетельствовать о том, что содержимое на сервере не изменилось с момента последнего обращения клиента. В этом случае метод возбудит исключение `NotModifiedException`, чтобы сообщить, что клиент должен извлечь данные из своего кеша.

12.4.3. Изменение ресурсов

Для выполнения операции PUT над ресурсом `RestTemplate` предлагает набор из трех простых методов `put()`. Как и все методы класса `RestTemplate`, метод `put()` имеет три версии:

```
void put(URI url, Object request) throws RestClientException;  
  
void put(String url, Object request, Object... uriVariables)  
    throws RestClientException;  
  
void put(String url, Object request, Map<String, ?> uriVariables)  
    throws RestClientException;
```

Простейшая версия метода `put()` принимает объект `java.net.URI`, идентифицирующий ресурс (и определяющий его местоположение), отправляемый на сервер, и Java-объект, представляющий этот ресурс.

Например, ниже показано, как можно использовать URI-версию метода `put()`, чтобы изменить ресурс `Spittle` на сервере:

```
public void updateSpittle(Spittle spittle) throws SpitterException {  
    try {  
        String url =  
            "http://localhost:8080/Spitter/spittles/" + spittle.getId();  
        new RestTemplate().put(new URI(url), spittle);  
    } catch (URISyntaxException e) {  
        throw new SpitterUpdateException("Unable to update Spittle", e);  
    }  
}
```

Несмотря на простую сигнатуру метода, возникают некоторые сложности с аргументом типа `java.net.URI`. Во-первых, чтобы создать URL изменяемого объекта `Spittle`, необходимо использовать операцию конкатенации строк. Во-вторых, поскольку есть вероятность сконструировать недопустимый URI, который затем передается конструктору класса `URI`, приходится перехватывать исключение `URISyntaxException` (даже если мы совершенно уверены в допустимости URI).

Другие версии метода `put()`, основанные на использовании строк, избавляют от неудобств, связанных с созданием экземпляра `URI`, включая необходимость перехватывать исключения. Более того, эти версии метода позволяют определять URI в виде шаблона, позволяя передавать переменные части шаблона. Ниже представлена измененная версия метода `updateSpittle()`, использующая один из методов `put()`, основанных на применении строк:

```
public void updateSpittle(Spittle spittle) throws SpitterException {
    restTemplate.put("http://localhost:8080/Spitter/spittles/{id}",
                    spittle, spittle.getId());
}
```

Теперь URI определяется как простой строковый шаблон. Когда `RestTemplate` отправляет PUT-запрос, вместо `{id}` в шаблон URI будет подставлено значение, возвращаемое методом `spittle.getId()`. Подобно методам `getForObject()` и `getForEntity()`, последний аргумент этой версии метода `put()` является списком аргументов переменной длины, замещающих переменные-заполнители в шаблоне в порядке их следования.

При желании значения для переменных-заполнителей в шаблонах можно передавать в виде отображения `Map`:

```
public void updateSpittle(Spittle spittle) throws SpitterException {
    Map<String, String> params = new HashMap<String, String>();
    params.put("id", spittle.getId());
    restTemplate.put("http://localhost:8080/Spitter/spittles/{id}",
                    spittle, params);
}
```

При использовании отображения для передачи переменных в шаблон, ключи элементов в отображении должны соответствовать именам переменных-заполнителей в шаблоне URI.

Во всех версиях `put()` во втором аргументе передается Java-объект, представляющий изменяемый ресурс, который будет отправлен на сервер методом `PUT`. В данном случае это объект `Spittle`. Для преобразования объектов `Spittle` перед отправкой на сервер класс `RestTemplate` будет использовать преобразователи HTTP-сообщений, перечисленные в табл. 12.2.

Формат преобразования объекта в значительной степени зависит от типа объекта, переданного методу `put()`. Если передать методу строковое значение, он будет использовать `StringHttpMessageConverter`: значение будет записано непосредственно в тело запроса, а в качестве типа содержимого будет выбрано значение `text/plain`. Отображение типа `MultiValueMap<String, String>` будет записано в тело запроса в форме `application/x-www-form-urlencoded` с помощью `FormHttpMessageConverter`.

Поскольку в данном случае будут отправляться объекты `Spittle`, нам необходим преобразователь сообщений, способный преобразовывать произвольные объекты. Если в библиотеке классов приложения присутствует библиотека Jackson JSON, для записи объектов `Spittle` в формате `application/json` будет использоваться преобразователь `MappingJacksonHttpMessageConverter`. Если класс `Spittle` отметить аннотацией поддержки сериализации JAXB и поместить библиотеку JAXB в библиотеку классов приложения, тогда объекты `Spittle` будут записываться в тело запроса в формате `application/xml`.

12.4.4. Удаление ресурсов

Когда ресурс становится ненужным, его можно удалить вызовом метода `delete()` класса `RestTemplate`. Подобно методу `put()`, метод `delete()` имеет три версии, сигнатуры которых приводятся ниже:

```
void delete(String url, Object... uriVariables)
    throws RestClientException;

void delete(String url, Map<String, ?> uriVariables)
    throws RestClientException;

void delete(URI url) throws RestClientException;
```

Вне всяких сомнений, метод `delete()` является самым простым из всех методов класса `RestTemplate`. Единственное, что требуется передать ему, – это `URI` удаляемого ресурса. Например, чтобы удалить

объект Spittle с указанным идентификатором, достаточно вызвать метод `delete()`, как показано ниже:

```
public void deleteSpittle(long id) {  
    try {  
        restTemplate.delete(  
            new URI("http://localhost:8080/Spitter/spittles/" + id));  
    } catch (URISyntaxException wontHappen) { }  
}
```

Достаточно просто, но здесь снова приходится использовать операцию конкатенации строк, чтобы создать объект `URI`. При этом конструктор может возбудить контролируемое исключение `URISyntaxException`, что вынуждает нас перехватывать его. Поэтому попробуем воспользоваться более простой версией метода `delete()`, чтобы избавиться от неудобств:

```
public void deleteSpittle(long id) {  
    restTemplate.delete("http://localhost:8080/Spitter/spittles/{id}", id);  
}
```

Так намного лучше. Не находите?

Теперь, после знакомства с набором наиболее простых методов класса `RestTemplate`, перейдем к знакомству с более сложными методами, поддерживающими запросы HTTP `POST`.

12.4.5. Создание новых ресурсов

Взглянув еще раз на табл. 12.3, можно заметить, что класс `RestTemplate` включает три разных метода для выполнения `POST`-запросов. Если умножить это число на три (по количеству версий каждого метода), получится девять методов, посылающих данные на сервер методом `POST`.

Имена двух из этих методов покажутся вам знакомыми. Методы `postForObject()` и `postForEntity()` выполняют `POST`-запросы почти так же, как методы `getForObject()` и `getForEntity()` выполняют `GET`-запросы. Другой метод, `getForLocation()`, является уникальным в этом отношении.

Прием объектов в ответах на POST-запросы

Представим, что нам необходимо воспользоваться классом `RestTemplate`, чтобы отправить на сервер новый объект `Spitter`. По-

скольку это совершенно новый объект Spitter, он пока не известен серверу. Поэтому официально он не является ресурсом REST и не имеет собственного URL. Кроме того, идентификатор его не будет известен клиенту, пока он не будет создан на сервере.

Один из способов отправить новый ресурс на сервер заключается в использовании метода `postForObject()` класса `RestTemplate`. Метод `postForObject()` имеет три версии со следующими сигнатурами:

```
<T> T postForObject(URI url, Object request, Class<T> responseType)
    throws RestClientException;

<T> T postForObject(String url, Object request, Class<T> responseType,
    Object... uriVariables) throws RestClientException;

<T> T postForObject(String url, Object request, Class<T> responseType,
    Map<String, ?> uriVariables) throws RestClientException;
```

Во всех версиях в первом параметре передается адрес URL, куда должен быть отправлен ресурс, во втором параметре – отправляемый объект, в третьем – Java-тип объекта, который, как ожидается, должен быть возвращен обратно. В версиях, которые принимают URL в виде строки, четвертый параметр определяет значения переменных-заполнителей в шаблоне URL (либо в виде списка аргументов переменной длины, либо в виде отображения `Map`).

В приложении Spitter новые ресурсы `Spitter` должны отправляться по адресу <http://localhost:8080/Spitter/spitters>, где находится метод-обработчик, сохраняющий объекты. Поскольку этот URL не содержит переменных, можно использовать любую версию метода `postForObject()`. Но в интересах простоты и чтобы избежать необходимости перехватывать исключения, которые могут возбуждаться при конструировании нового объекта `URI`, мы реализуем эту операцию, как показано ниже:

```
public Spitter postSpitterForObject(Spitter spitter) {
    RestTemplate rest = new RestTemplate();
    return rest.postForObject("http://localhost:8080/Spitter/spitters",
        spitter, Spitter.class);
}
```

Метод `postSpitterForObject()` получает новый объект `Spitter` и с помощью `postForObject()` отправляет его на сервер. В ответ он получает объект `Spitter` и возвращает его вызывающей программе.

Как и при использовании метода `getForObject()`, может возникнуть потребность исследовать некоторые метаданные, поступающие вместе с ответом. В этом случае предпочтительнее будет использовать метод `postForEntity()`. Метод `postForEntity()` имеет три версии, сигнатуры которых являются почти точным отражением сигнатур версий метода `postForObject()`:

```
<T> ResponseEntity<T> postForEntity(URI url, Object request,
                                         Class<T> responseType) throws RestClientException;

<T> ResponseEntity<T> postForEntity(String url, Object request,
                                         Class<T> responseType, Object... uriVariables)
                                         throws RestClientException;

<T> ResponseEntity<T> postForEntity(String url, Object request,
                                         Class<T> responseType, Map<String, ?> uriVariables)
                                         throws RestClientException;
```

Предположим, что помимо возвращаемого обратно ресурса Spitter необходимо также извлечь из ответа значение заголовка `Location`. В этом случае можно воспользоваться методом `postForEntity()`, как показано ниже:

```
RestTemplate rest = new RestTemplate();
ResponseEntity<Spitter> response = rest.postForEntity(
    "http://localhost:8080/Spitter/spitters", spitter, Spitter.class);

Spitter spitter = response.getBody();
URI url = response.getHeaders().getLocation();
```

Подобно методу `getForEntity()`, метод `postForEntity()` возвращает объект `ResponseEntity<T>`. Извлечь ресурс (в данном случае – объект Spitter) из этого объекта можно с помощью его метода `getBody()`. А с помощью метода `getHeaders()` можно извлечь объект `HttpHeaders` и уже с его помощью получить доступ к различным HTTP-заголовкам ответа. В примере выше значение заголовка `Location` извлекается вызовом метода `getLocation()`, который возвращает его в виде объекта `java.net.URI`.

Прием местоположения ресурса после выполнения POST-запроса

Метод `postForEntity()` удобно использовать для приема отправленного ресурса и заголовков ответа. Но часто бывает так, что нет

необходимости принимать отправленный ресурс обратно (в конце концов, этот ресурс уже имеется в приложении). Если значение заголовка `Location` – это все, что представляет интерес, тогда проще будет воспользоваться методом `postForLocation()` класса `RestTemplate`.

Подобно другим методам, выполняющим POST-запросы, метод `postForLocation()` отправляет ресурс на сервер в теле POST-запроса. Но вместо объекта ресурса метод `postForLocation()` возвращает местоположение вновь созданного ресурса. Три версии этого метода имеют следующие сигнатуры:

```
URI postForLocation(String url, Object request, Object... uriVariables)
    throws RestClientException;

URI postForLocation(
    String url, Object request, Map<String, ?> uriVariables)
    throws RestClientException;

URI postForLocation(URI url, Object request) throws RestClientException;
```

Для демонстрации метода `postForLocation()` попробуем реализовать отправку объекта `Spitter` еще раз. На этот раз потребуем вернуть URL ресурса:

```
public String postSpitter(Spitter spitter) {
    RestTemplate rest = new RestTemplate();
    return rest.postForLocation("http://localhost:8080/Spitter/spitters",
        spitter).toString();
}
```

Здесь адрес URL передается в виде строки (в данном случае шаблон URL не содержит переменных-заполнителей). Если после создания ресурса сервер вернет его URL в заголовке `Location` ответа, то метод `postForLocation()` вернет этот URL в виде строки.

12.4.6. Обмен ресурсами

К настоящему моменту мы познакомились со всеми основными методами класса `RestTemplate`, которые используются для получения, изменения, удаления и создания новых ресурсов. Наряду с ними мы рассмотрели также два специальных метода, `getForEntity()` и `postForEntity()`, возвращающие ресурс, обернутый объектом `RequestEntity`, из которого можно извлечь код состояния HTTP и заголовки ответа.

Возможность чтения заголовков ответа весьма полезна на практике. Но что, если приложению потребуется отправлять на сервер заголовки запроса? Для этой цели можно использовать метод `exchange()`.

Подобно остальным методам класса `RestTemplate`, метод `exchange()` имеет три перегруженные версии со следующими сигнатурами:

```
<T> ResponseEntity<T> exchange(URI url, HttpMethod method,
    HttpEntity<?> requestEntity, Class<T> responseType)
    throws RestClientException;

<T> ResponseEntity<T> exchange(String url, HttpMethod method,
    HttpEntity<?> requestEntity, Class<T> responseType,
    Object... uriVariables) throws RestClientException;

<T> ResponseEntity<T> exchange(String url, HttpMethod method,
    HttpEntity<?> requestEntity, Class<T> responseType,
    Map<String, ?> uriVariables) throws RestClientException;
```

Как видите, сигнатуры трех перегруженных версий метода `exchange()` соответствуют шаблону, общему для всех методов класса `RestTemplate`. Первая версия принимает объект `java.net.URI`, идентифицирующий целевой URL, а остальные две принимают URL в виде строки и аргументы со значениями для переменных-заполнителей в URL.

Метод `exchange()` принимает также параметр `HttpMethod`, который определяет, какой метод HTTP должен использоваться. В зависимости от значения этого параметра метод `exchange()` может выполнять те же операции, что и любые другие методы класса `RestTemplate`.

Например, как показано ниже, извлечь ресурс `Spitter` можно с помощью метода `getForEntity()`:

```
ResponseEntity<Spitter> response = rest.getForEntity(
    "http://localhost:8080/Spitter/spitters/{spitter}",
    Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

А также с помощью метода `exchange()`:

```
ResponseEntity<Spitter> response = rest.exchange(
    "http://localhost:8080/Spitter/spitters/{spitter}",
    HttpMethod.GET, null, Spitter.class, spitterId);
Spitter spitter = response.getBody();
```



В этом примере, передав значение `HttpMethod.GET` в качестве метода HTTP, приложение предлагает методу `exchange()` отправить GET-запрос. Третий аргумент предназначен для отправляемого ресурса, но, поскольку в данном случае выполняется запрос GET, в нем передается пустая ссылка (`null`). Следующий аргумент определяет ожидаемый тип возвращаемого объекта. И последний аргумент – это значение, которое будет подставлено на место переменной-заполнителя `{spitter}` в указанном шаблоне URL.

При таком использовании метод `exchange()` практически идентичен методу `getForEntity()`. Но, в отличие от `getForEntity()` или `getForObject()`, метод `exchange()` позволяет устанавливать заголовки в запросе. Вместо значения `null` методу `exchange()` можно передать объект `HttpEntity` с требуемыми заголовками.

Если приложение не определяет заголовки, метод `exchange()` отправит GET-запрос на получение ресурса `Spitter` со следующими заголовками:

```
GET /Spitter/spitters/habuma HTTP/1.1
Accept: application/xml, text/xml, application/*+xml, application/json
Content-Length: 0
User-Agent: Java/1.6.0_20
Host: localhost:8080
Connection: keep-alive
```

Обратите внимание на заголовок `Accept`. Он сообщает серверу, что приложение способно принимать содержимое в нескольких форматах XML, а также в формате `application/json`. Это дает серверу большую свободу выбора при определении формата представления возвращаемого ресурса. Предположим, что необходимо ограничить сервер форматом JSON. В этом случае следует оставить в заголовке `Accept` только значение `application/json`.

Установка заголовков запроса сводится к созданию объекта `HttpEntity`, содержащего объект `MultiValueMap` с требуемыми заголовками:

```
MultiValueMap<String, String> headers =
    new LinkedMultiValueMap<String, String>();
headers.add("Accept", "application/json");
HttpEntity<Object> requestEntity = new HttpEntity<Object>(headers);
```

Здесь создается объект `LinkedMultiValueMap` и в него добавляется заголовок `Accept` со значением `application/json`. Затем создается объект

HttpEntity (с обобщенным типом Object) вызовом конструктора, которому передается аргумент с объектом MultiValueMap. Если бы это был запрос PUT или POST, точно так же можно было бы добавить в объект HttpEntity тело запроса – для запроса GET в этом нет необходимости.

Теперь можно вызвать метод exchange(), передав ему объект HttpEntity:

```
ResponseEntity<Spitter> response = rest.exchange(
    "http://localhost:8080/Spitter/spitters/{spitter}",
    HttpMethod.GET, requestEntity, Spitter.class, spitterId);
Spitter spitter = response.getBody();
```

На первый взгляд кажется, что результат будет тот же самый. Приложение должно получить запрошенный объект Spitter. Но в действительности на сервер будет отправлен запрос со следующими заголовками:

```
GET /Spitter/spitters/habuma HTTP/1.1
Accept: application/json
Content-Length: 0
User-Agent: Java/1.6.0_20
Host: localhost:8080
Connection: keep-alive
```

И если сервер способен преобразовать объект Spitter в формат JSON, тело ответа должно быть представлено в формате JSON.

В этом разделе было показано, как пользоваться различными методами класса RestTemplate и как с их помощью приложения-клиенты, написанные на языке Java, могут взаимодействовать с ресурсами RESTful на сервере. Но что, если клиентом является веб-браузер? Когда доступ к ресурсам REST осуществляется с помощью браузера, необходимо учитывать некоторые ограничения, особенно это касается диапазона методов HTTP, поддерживаемых браузером. В завершение этой главы посмотрим, как фреймворк Spring помогает преодолевать данные ограничения.

12.5. Отправка форм в стиле RESTful

Мы познакомились с четырьмя основными методами протокола HTTP – GET, POST, PUT и DELETE, – определяющими основные операции, которые можно выполнять над ресурсами. И теперь знаем, как установкой атрибута method аннотации @RequestMapping заставить

DispatcherServlet передавать HTTP-запросы определенным методам контроллеров. Фреймворк Spring MVC способен обрабатывать любые типы HTTP-запросов, посылаемые клиентом.

Основные проблемы в этом отношении связаны с HTML и веб-браузерами. Клиенты, не являющиеся браузерами, такие как приложения, использующие класс RestTemplate, не должны испытывать проблем в выполнении HTTP-запросов любых типов. Но стандарт HTML 4 официально поддерживает только методы GET и POST, оставляя в стороне PUT, DELETE и все остальные методы протокола HTTP. Даже при том, что стандарт HTML 5 и новейшие версии браузеров поддерживают все методы HTTP, вы едва ли можете рассчитывать, что пользователи вашего приложения будут использовать современные браузеры.

Обычно, чтобы обойти недостатки стандарта HTML 4 и старых браузеров, используется прием подмены запросов PUT и DELETE запросами POST. Он заключается в передаче запроса POST со скрытым полем, несущим в себе фактическое имя метода HTTP. Когда запрос поступает на сервер, его тип переопределяется в соответствии со значением скрытого поля.

Фреймворк Spring поддерживает прием подмены запросов с помощью следующих двух особенностей:

- ❑ преобразование типа запроса с помощью `HiddenHttpMethodFilter`;
- ❑ отображение скрытого поля с помощью JSP-тега `<sf:form>`.

Посмотрим сначала, как использовать тег `<sf:form>` для отображения скрытого поля с целью подмены типа запроса.

12.5.1. Отображение скрытого поля с именем метода

В разделе 8.4.1 было показано, как использовать библиотеку связывания полей формы, входящую в состав Spring, для отображения HTML-форм. Основным элементом этой библиотеки является тег `<sf:form>`. Этот тег определяет содержимое для других тегов, связывающих поля формы с атрибутами модели.

Мы уже использовали тег `<sf:form>` для определения формы создания нового объекта Spitter. Для этого запрос POST подходит как нельзя лучше, потому что этот тип запросов часто используется для создания новых ресурсов. Но как быть, если потребуется изменить или удалить ресурс? Для таких ситуаций лучше подходят запросы PUT и DELETE соответственно.

Однако, как уже отмечалось выше, HTML-тег `<form>` не всегда может посыпать иные запросы, кроме GET и POST. Некоторые новейшие браузеры не испытывают проблем, если в атрибуте `method` тега `<form>` указать метод PUT или DELETE, но более старые браузеры могут скрыто посыпать формы серверу методом POST.

Чтобы обеспечить подмену типа POST запроса на PUT или DELETE, необходимо создать HTML-форму, отправляемую серверу в виде POST-запроса, и добавить в нее скрытое поле. Например, следующий фрагмент разметки HTML демонстрирует, как создать форму, которая отправляется в виде запроса DELETE:

```
<form method="post">
    <input type="hidden" name="_method" value="delete"/>
    ...
</form>
```

Как видите, совсем несложно создать форму со скрытым полем, определяющим настоящий метод HTTP. Для этого достаточно добавить скрытое поле с именем, которое будет опознано сервером, и указать в этом поле имя желаемого метода HTTP. Эта форма будет отправлена на сервер в виде POST-запроса. А сервер, по всей видимости, будет интерпретировать поле `_method` как фактический тип запроса (чуть ниже будет показано, как настроить сервер, чтобы обеспечить эту интерпретацию).

С помощью тега `<sf:form>` из библиотеки связывания форм, входящей в состав фреймворка Spring, сделать это еще проще. Нужно лишь указать в его атрибуте `method` желаемый метод HTTP, и тег `<sf:form>` сам позаботится о создании скрытого поля:

```
<sf:form method="delete" modelAttribute="spitter">
    ...
</sf:form>
```

При отображении тега `<sf:form>` в разметку HTML результат будет очень похож на HTML-тег `<form>`, показанный выше. Тег `<sf:form>` освобождает от необходимости вручную определять скрытое поле, позволяя определять формы, отправляемые в виде запросов PUT и DELETE, более естественным способом, как если бы они поддерживались браузерами.

Но тег `<sf:form>` – это лишь часть механизма подмены типов запросов, действующая на стороне браузера. А как сервер узнает, что POST-запрос должен обрабатываться как запрос PUT или DELETE?

12.5.2. Преобразование типа запроса

Когда браузер посыпает запрос PUT или DELETE, отправляя форму, отображенную с помощью тега `<sf:form>`, в действительности он выполняет запрос методом POST. Через сеть запрос проходит как POST-запрос, прибывает на сервер как POST-запрос, и если приложение на стороне сервера не потрудится заглянуть в скрытое поле `_method`, он будет обработан как POST-запрос.

Однако в нашем контроллере имеются методы-обработчики, отмеченные аннотацией `@RequestMapping`, которые предназначены для обработки запросов PUT и DELETE. Поэтому, прежде чем `DispatcherServlet` попытается определить метод-обработчик для передачи запроса, некоторый механизм должен разрешить проблему несоответствия HTTP-метода. Эту работу в фреймворке Spring выполняет `HiddenHttpMethodFilter`.

`HiddenHttpMethodFilter` – это сервлет-фильтр и настраивается в файле `web.xml`, как показано ниже:

```
<filter>
    <filter-name>httpMethodFilter</filter-name>
    <filter-class>
        org.springframework.web.filter.HiddenHttpMethodFilter
    </filter-class>
</filter>
...
<filter-mapping>
    <filter-name>httpMethodFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

Здесь сервлет `HiddenHttpMethodFilter` отображается на шаблон URL `/*`. Поэтому, прежде чем попасть в `DispatcherServlet`, все запросы будут проходить фильтр `HiddenHttpMethodFilter`.

Как показано на рис. 12.3, сервлет `HiddenHttpMethodFilter` преобразует запросы PUT и DELETE, полученные в виде POST-запросов, в их истинную форму. Когда POST-запрос достигает сервера, сервлет `HiddenHttpMethodFilter` проверяет поле `_method` и в соответствии с его значением устанавливает фактический метод HTTP.

К тому моменту, когда запрос достигнет `DispatcherServlet` и метода контроллера, он уже будет преобразован. Никто не сможет догадаться, что запрос был рожден как POST-запрос. Благодаря способности тега `<sf:form>` автоматически отображать скрытое поле и спо-

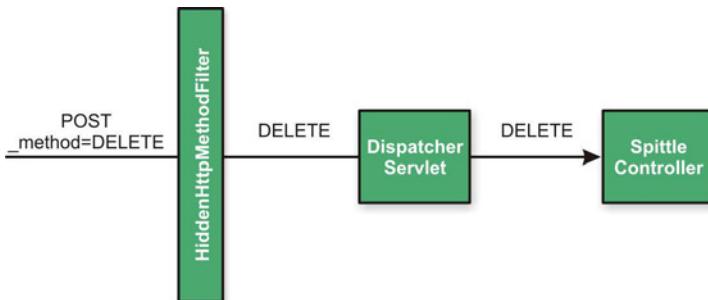


Рис. 12.3. Сервлет HiddenHttpMethodFilter преобразует запросы PUT и DELETE, замаскированные под POST-запросы, в их истинную форму

собности HiddenHttpMethodFilter преобразовывать запросы, опираясь на значение этого скрытого поля, вам не придется волноваться об обработке типов HTTP-запросов, не поддерживаемых браузером, при создании JSP-форм и контроллеров Spring MVC.

Прежде чем оставить тему подмены POST-запросов, я хочу напомнить, что этот прием разрабатывался лишь как обходное решение для поддержки запросов PUT и DELETE в старых браузерах. Другие приложения, включая те, что используют класс RestTemplate, не должны испытывать проблем с отправкой HTTP-запросов любых типов. То есть если приложение не будет обрабатывать запросы PUT и DELETE, получаемые от браузеров, тогда нет необходимости пользоваться услугами HiddenHttpMethodFilter.

12.6. В заключение

Архитектура RESTful позволяет интегрировать приложения, основываясь на веб-стандартах, сохраняя взаимодействия простыми и естественными. Ресурсы в системе идентифицируются адресами URL, управляются с помощью методов HTTP и отображаются в форму, наиболее соответствующую требованиям клиента.

В этой главе вы узнали, как писать контроллеры Spring MVC, обрабатывающие запросы управления ресурсами RESTful. Используя шаблоны параметризованных адресов URL и связывая методы контроллеров с конкретными методами HTTP, можно обеспечить обработку запросов GET, POST, PUT и DELETE на выполнение операций с ресурсами приложения.

В ответ на запросы фреймворк Spring может обеспечить представление данных из ресурсов в формате, наиболее соответствующем требованиям клиента. При использовании представлений можно воспользоваться услугами арбитра представлений `Content-NegotiatingViewResolver`, который выберет представление из числа предлагаемых другими арбитрами представлений, лучше всего удовлетворяющее потребности клиента. Кроме того, метод-обработчик контроллера можно отметить аннотацией `@ResponseBody`, избавившись тем самым от этапа выбора представления, и поручить одному из нескольких преобразователей HTTP-сообщений превратить возвращаемое значение в ответ, который будет отправлен клиенту.

Для поддержки диалога в стиле REST на стороне клиента фреймворк Spring предоставляет класс `RestTemplate`, обеспечивающий механизм взаимодействия с ресурсами RESTful на основе шаблонов. А если клиентом является браузер, отсутствие поддержки HTTP-методов `PUT` и `DELETE` может быть восполнено с помощью сервлет-фильтра `HiddenHttpMethodFilter`.

Несмотря на существенные отличия взаимодействий в архитектуре RESTful, описанных в этой главе, от диалогов в стиле RPC, описанных в предыдущей главе, они имеют одну общую черту: по своей природе они являются синхронными. Когда клиент отправляет сообщение на сервер, он предполагает, что сервер будет готов вернуть ответ немедленно. В противоположность этому асинхронные взаимодействия позволяют серверу реагировать на сообщение по мере возможности и необязательно мгновенно. В следующей главе будет показано, как с помощью Spring обеспечить асинхронную интеграцию приложений.



Глава 13. Обмен сообщениями в Spring

В этой главе рассматриваются следующие темы:

- ❑ введение в Java Message Service (JMS);
- ❑ прием и передача асинхронных сообщений;
- ❑ объекты POJO, управляемые сообщениями.

Пятница, 16:55. Вы в нескольких минутах от начала долгожданного отпуска. Времени до вылета самолета в обрез, только чтобы доехать до аэропорта. Но прежде необходимо убедиться, что ваши коллеги и начальник будут знать, в каком месте вы закончили работу, чтобы продолжить ее в понедельник. К сожалению, некоторые коллеги ушли чуть раньше, а начальник занят на совещании. Что делать?

Можно было бы позвонить начальнику на сотовый телефон.., но нельзя прерывать совещание, только чтобы доложить о состоянии дел. Можно было бы подождать, пока он не вернется в кабинет. Но нельзя предугадать, как долго продлится совещание, да и до самолета времени в обрез. Можно было бы приkleить бумажку на монитор.., но она легко затеряется среди сотни других таких же.

Самый простой выход отчитаться о состоянии дел и успеть на самолет – послать электронное письмо начальнику и коллегам, заодно пообещав прислать открытку из отпуска. Хотя вы и не знаете, когда они прочтут свою электронную почту, зато знаете наверняка, что рано или поздно они вернутся к своим компьютерам и смогут прочитать сообщение. А вы тем временем уже будете мчаться к аэропорту.

Иногда прямой контакт с собеседником просто необходим. Если вы поранитесь и вам потребуется медицинская помощь, вы наверняка приметесь звонить по телефону – отправка электронного письма в больницу в этой ситуации не поможет. Но часто бывает достаточно просто отправить сообщение, например чтобы получить разрешение продолжить свой отпуск.

В двух предыдущих главах рассказывалось, как пользоваться механизмами RMI, Hessian, Burlap, HTTP Invoker и веб-службами для организации взаимодействий между приложениями. Все эти механизмы взаимодействий имеют синхронную природу, когда клиентское приложение направляет запрос на удаленную службу и ожидает, пока удаленная процедура не выполнится, прежде чем продолжить работу.

Синхронные взаимодействия имеют место быть, но это не единственный способ организации взаимодействий между приложениями. Асинхронный обмен сообщениями позволяет отправлять сообщения из одного приложения в другое, не ожидая ответа. *Асинхронный способ взаимодействий* имеет свои преимущества перед синхронным, как будет показано ниже.

Java Message Service (JMS) – стандартный API для организации асинхронного обмена сообщениями. В этой главе будет показано, насколько фреймворк Spring упрощает отправку и прием сообщений с помощью JMS. Помимо простых операций приема и передачи сообщений, здесь также будет рассказываться о простых Java-объектах (POJO), управляемых сообщениями, как способе приема сообщений, напоминающем прием сообщений компонентами EJB, управляемыми сообщениями (Message-Driven Beans, MDB).

13.1. Краткое введение в JMS

JMS – это механизм организации взаимодействий между приложениями, во многом напоминающий механизмы удаленных взаимодействий и интерфейсы REST, представленные в предыдущих главах. Но важным отличием JMS от других механизмов является способ передачи данных между системами.

Механизмы удаленных взаимодействий, такие как RMI и Hessian/Burlap, имеют синхронную природу. Как показано на рис. 13.1, когда клиент вызывает удаленный метод, он вынужден ждать завершения выполнения метода, прежде чем продолжить работу. Даже если удаленный метод ничего не возвращает клиенту, клиент все равно будет простаивать, пока обслуживание его запроса не завершится.

Механизм JMS, напротив, обеспечивает асинхронный режим взаимодействий между приложениями. Когда сообщения отправляются асинхронно, как показано на рис. 13.2, клиенту не приходится ждать, пока служба обработает сообщение или доставит его адресату. Клиент просто отправляет свое сообщение и продолжает работу, надеясь, что служба рано ли поздно примет и обработает его.

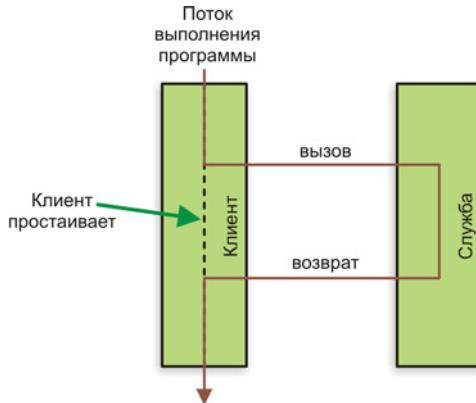


Рис. 13.1. При синхронных взаимодействиях клиент вынужден ждать завершения операции

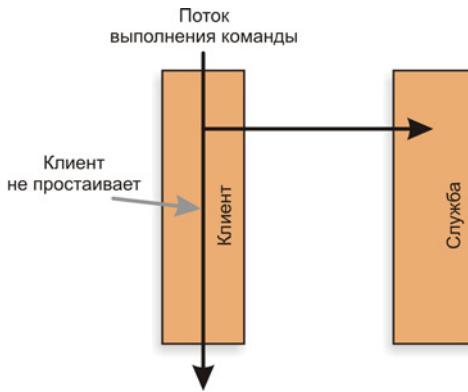


Рис. 13.2. При асинхронных взаимодействиях клиенту не приходится ждать ответа службы

Асинхронные взаимодействия посредством JMS имеют ряд преимуществ перед синхронными взаимодействиями. Эти преимущества будут рассматриваться чуть ниже. Но сначала посмотрим, как выполняется отправка сообщений с помощью JMS.

13.1.1. Архитектура JMS

Большинство из нас не раз пользовались услугами почты. Ежедневно миллионы людей передают письма, открытки и посылки в руки почтальонов, будучи уверенными, что они будут доставлены



адресатам. Мир слишком велик, чтобы все эти отправления можно было передавать из рук в руки лично, поэтому мы полагаемся на почтовую службу. Мы обращаемся на почту, оплачиваем услуги за пересылку, передаем почтовое отправление и ни на секунду не задумываемся, каким образом оно будет доставлено.

Преимуществом почтовой службы является отсутствие необходимости прямого участия в доставке. Было бы крайне неудобно лично заниматься доставкой, например, поздравительной открытки бабушке на ее день рождения. В зависимости от расстояния, разделяющего вас, чтобы доставить открытку лично, пришлось бы отложить свои дела на несколько часов, а то и дней. К счастью, почта доставит эту открытку без вашего участия.

Аналогично действует и механизм JMS. Когда одно приложение отправляет информацию другому приложению посредством JMS, не требуется устанавливать прямую связь между двумя приложениями. Вместо этого приложение-отправитель передает сообщение в руки службе, которая гарантирует его доставку приложению-получателю.

В JMS имеются две основные действующие стороны: *брокеры сообщений* и *приемники*.

Когда приложение отправляет сообщение, оно передается в руки брокера сообщений. Брокер сообщений – это аналог почтового отделения в JMS. Брокер сообщений гарантирует доставку сообщения в указанный приемник, позволяя отправителю продолжать заниматься своими делами.

При отправке письма по почте важно указать адрес на конверте, чтобы работники почтовой службы могли узнать, куда его доставить. В JMS также имеется понятие адресов, которые определяют приемники. Приемники – это своего рода почтовые ящики, куда будут помещаться сообщения.

Но, в отличие от почтовых адресов, которые определяют имя человека или номер дома с названием улицы, адреса в JMS являются



Рис. 13.3. Очередь сообщений обеспечивает независимость отправителя сообщения от его получателя.

Несмотря на то что очередь может использоваться несколькими получателями, каждое сообщение может быть вынуто из очереди только один раз

менее определенными. Адреса в JMS определяют лишь место, где получатель сможет получить сообщение, но не определяют, *кто* сможет получить его. То есть в JMS допускается посыпать письма по адресу, например: «действующему Президенту».

В JMS существуют два типа приемников: *очереди* и *темы*. Каждый из них следует определенной модели обмена сообщениями: либо модель «точка–точка» (очереди), либо модель «публикация–подписка» (темы).

Модель «точка–точка»

В модели «точка–точка» каждое сообщение имеет точно одного получателя и одного отправителя, как показано на рис. 13.3. Когда брокер сообщений получает сообщение, он помещает его в очередь. Когда получатель обращается за следующим сообщением в очереди, это сообщение удаляется из очереди и передается получателю. Поскольку в момент доставки сообщение удаляется из очереди, его гарантированно получит только один получатель.

Несмотря на то что каждое сообщение, имеющееся в очереди, передается только одному получателю, это не означает, что очередью может пользоваться только один получатель. В действительности сообщения из очереди могут извлекаться несколькими получателями. Но каждый из них получит свое, уникальное сообщение.

Это напоминает ожидание в очереди в банке. В процессе ожидания можно заметить, что очередь обслуживается несколькими операторами банка. Как только один клиент будет обслужен, оператор вызовет следующего из очереди. Когда подойдет ваша очередь, вас вызовут, и вы будете обслужены одним из операторов. Другие операторы будут обслуживать других клиентов.

Еще одно наблюдение, которое можно сделать в банке, – когда клиент становится в очередь, он не знает наверняка, какой оператор будет его обслуживать. Можно прикинуть количество людей в очереди, соотнести это число с количеством операторов, учесть, кто из операторов работает быстрее других, и затем выдвинуть предположение о том, кто из операторов будет вас обслуживать. Но велика вероятность, что вы ошибетесь, и вас будет обслуживать другой оператор.

То же самое происходит и в JMS, если очередь обслуживается несколькими получателями, нет никакой возможности определить, который из них будет обрабатывать то или иное сообщение. Эта неопределенность – благо, потому что позволяет приложению увели-

чивать скорость обработки сообщений простым добавлением новых получателей.

Модель «публикация–подписка»

В модели «публикация–подписка» сообщения отправляются в тему. Подобно очередям, тему могут обслуживать несколько получателей. Но, в отличие от очередей, где каждое сообщение передается точно одному получателю, все подписчики на тему получат собственные копии сообщений, как показано на рис. 13.4.

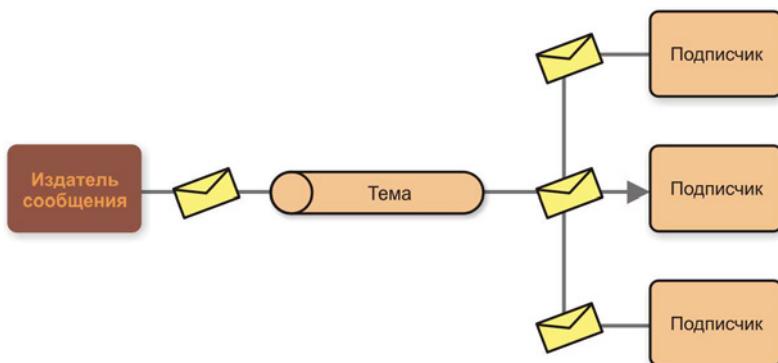


Рис. 13.4. Подобно очередям, темы обеспечивают независимость отправителей сообщений от их получателей. В отличие от очередей, сообщение в теме может быть доставлено нескольким получателям, подписавшимся на тему

Как следует из названия модели «публикация–подписка», она во многом напоминает модель издательства, выпускающего журнал, и подписчиков. Журнал (сообщение) публикуется и передается почтовой службе, которая затем доставляет копии журнала подписчикам.

Аналогия с журналом не совсем корректна в данном случае, потому что в JMS издатель не знает, кто будет играть роль подписчика. Единственное, что известно издателю, – его сообщения будут опубликованы в определенной теме, но не известно, кто подписан на эту тему. Из этого также следует, что издатель не знает, как будут обрабатываться его сообщения.

Теперь, после знакомства с основами JMS, можно попытаться сравнить асинхронный механизм JMS с синхронным механизмом RPC.

13.1.2. Преимущества JMS

Несмотря на всю свою простоту, синхронные взаимодействия налагают некоторые ограничения на удаленные службы и их клиентов, наиболее значимые из которых перечислены ниже.

- ❑ Синхронные взаимодействия предполагают наличие этапов ожидания. Когда клиент вызывает метод удаленной службы, он вынужден ожидать завершения удаленного метода, прежде чем продолжить работу. Если клиент обращается к удаленной службе достаточно часто и/или служба имеет невысокое быстродействие, это может отрицательно сказаться на производительности клиентского приложения.
- ❑ Клиент тесно связан со службой через интерфейс службы. Если потребуется изменить интерфейс службы, придется изменить и все клиентские приложения.
- ❑ Клиент тесно связан с местонахождением службы. Клиента необходимо настраивать, чтобы сообщить ему сетевой адрес службы. Если топология сети изменится, клиентов придется перенастраивать, чтобы указать им новый адрес службы.
- ❑ Клиент существенно зависит от доступности службы. Если служба окажется недоступной, клиент окажется парализованным.

Синхронные взаимодействия имеют право на существование, но перечисленные выше недостатки необходимо учитывать при выборе механизма взаимодействий для своего приложения. Если перечисленные ограничения не устраивают вас, подумайте о возможности использования асинхронных взаимодействий с помощью механизма JMS, свободного от этих проблем.

Отсутствие ожидания

Когда передача сообщений осуществляется с помощью JMS, клиент не должен ждать, пока оно будет обработано или хотя бы доставлено. Клиент отдает сообщение брокеру и продолжает свое выполнение, надеясь, что сообщение будет доставлено по указанному адресу.

Поскольку клиенту не приходится ждать, он может заниматься решением других задач. Отсутствие простого может существенно повысить производительность клиента.

Ориентированность на сообщения и независимость

В отличие от механизма RPC, суть которого состоит в организации вызовов методов, в основе механизма JMS лежат данные. То



есть клиент не стеснен рамками сигнатуры метода. Он сможет поместить в очередь или в тему любое сообщение, и ему не требуется знать все тонкости его обработки.

Независимость от местоположения

Синхронные службы RPC обычно связаны с определенными сетевыми адресами. Как следствие, клиенты оказываются очень чувствительны к изменениям топологии сети. Если IP-адрес или порт службы изменится, потребуется изменить соответствующие настройки клиентов, иначе они не смогут получить доступ к службе.

Клиенты JMS, напротив, понятия не имеют, кто будет обрабатывать их сообщения или где находится служба. Клиент знает только очередь или тему для отправки сообщений. В результате клиент оказывается независимым от службы, при условии что он в состоянии извлекать сообщения из очереди или из темы.

Преимущество независимости от местоположения службы в модели «точка–точка» можно использовать для создания кластеров служб. Если клиенту не требуется знать местоположение службы и единственным требованием службы является доступность брокера сообщений, то нет никаких причин, почему нельзя было бы настроить несколько служб на извлечение сообщений из единой очереди. Если службы оказываются перегруженными и начинают запаздывать, достаточно просто добавить еще несколько экземпляров службы, обслуживающих ту же очередь.

Независимость от местоположения влечет за собой еще один интересный побочный эффект в модели «публикация–подписка». На одну тему может быть подписано несколько служб, извлекающих копии одного и того же сообщения. Но каждая служба обрабатывает сообщение по-своему. Например, допустим, что имеется множество служб, реализующих обработку информации при найме нового работника. Одна из служб может добавлять работника в систему расчета заработной платы, другая – в систему учета персонала, а третья – открывает работнику доступ к системам, которые ему потребуются для выполнения своих служебных обязанностей. Каждая служба действует независимо и использует те же исходные данные, что и другие службы, подписанные на получение сообщений из темы.

Гарантированная доставка

Чтобы клиент мог взаимодействовать с синхронной службой, служба должна быть готова к приему запросов на указанном IP-

адресе с указанным номером порта. Если служба не будет запущена или будет недоступна по каким-то другим причинам, клиент не сможет продолжить выполнение.

Когда отправка сообщений производится посредством JMS, клиент может быть уверен, что его сообщения будут доставлены службе. Даже если служба окажется недоступна на момент отправки сообщения, оно будет храниться, пока служба снова не станет доступна.

Теперь, когда вы получили общее представление о JMS и асинхронной передаче сообщений, попробуем настроить брокер сообщений JMS, который будет использоваться в наших примерах. Вы можете использовать любой брокер сообщений JMS, но здесь будет использоваться наиболее популярный брокер *ActiveMQ*.

13.2. Настройка брокера сообщений в Spring

ActiveMQ – отличный брокер сообщений, распространяемый с открытыми исходными текстами, и замечательный выбор для организации асинхронного обмена сообщениями посредством JMS. На момент написания этих строк текущей была версия ActiveMQ 5.4.2. Прежде чем приступить к работе с ActiveMQ, необходимо получить дистрибутив, который можно загрузить по адресу: <http://activemq.apache.org>. После загрузки дистрибутива распакуйте его в каталог на жестком диске. В подкаталоге `lib` распакованного дистрибутива отыщите файл `activemqcore-5.4.2.jar`. Этот файл необходимо добавить в библиотеку классов (`classpath`) приложения, чтобы получить возможность пользоваться ActiveMQ API.

В каталоге `bin` имеются несколько подкаталогов, предназначенных для различных операционных систем. Внутри этих подкаталогов находятся сценарии запуска ActiveMQ. Например, чтобы запустить ActiveMQ в Mac OS X, выполните сценарий `activemq` из каталога `bin/macosx`. Через мгновение после этого ActiveMQ будет готов к приему сообщений.

13.2.1. Создание фабрики соединений

В этой главе будут рассматриваться различные способы отправки и приема сообщений с помощью фреймворка Spring. Во всех случаях, чтобы иметь возможность отправлять сообщения через брокер, нам потребуется использовать фабрику соединений JMS. Поскольку

в этой главе предполагается использовать брокер ActiveMQ, необходимо настроить фабрику соединений JMS так, чтобы она создавала соединения с ActiveMQ. В состав ActiveMQ входит собственная фабрика соединений ActiveMQConnectionFactory, которая настраивается в Spring, как показано ниже:

```
<bean id="connectionFactory"
      class="org.apache.activemq.spring.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

Кроме того, поскольку известно, что будет использоваться брокер ActiveMQ, для объявления фабрики соединений можно использовать пространство имен, определяемое брокером ActiveMQ (доступно во всех версиях ActiveMQ, начиная с версии 4.1). Сначала необходимо подключить пространство имен amq в конфигурационном XML-файле Spring:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jms="http://www.springframework.org/schema/jms"
       xmlns:amq="http://activemq.apache.org/schema/core"
       xsi:schemaLocation="http://activemq.apache.org/schema/core
                           http://activemq.apache.org/schema/core/activemq-core-5.5.0.xsd
                           http://www.springframework.org/schema/jms
                           http://www.springframework.org/schema/jms/spring-jms-3.0.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    ...
</beans>
```

А затем с помощью элемента `<amq:connectionFactory>` объявить фабрику соединений:

```
<amq:connectionFactory id="connectionFactory"
                       brokerURL="tcp://localhost:61616"/>
```

Обратите внимание, что элемент `<amq:connectionFactory>` является особенностью брокера ActiveMQ. При использовании других реализаций брокера сообщений конфигурационные пространства имен могут отсутствовать, и в этом случае необходимо объявлять фабрику соединений как обычный компонент.

Далее в этой главе мы часто будем возвращаться к определению компонента `connectionFactory`. Но сейчас достаточно будет знать, что атрибут `brokerURL` определяет местоположение брокера. В данном случае URL в атрибуте `brokerURL` сообщает, что брокер действует на локальном компьютере и принимает соединения на порту с номером 61616 (номер порта, используемый брокером ActiveMQ по умолчанию).

13.2.2. Объявление приемников ActiveMQ

Помимо фабрики соединений, необходимо настроить приемник, откуда можно будет извлекать сообщения. Приемник может быть либо очередью, либо темой, в зависимости от потребностей приложения.

Независимо от типа приемника, очередь или тема, компонент приемник должен быть настроен как экземпляр класса, характерного для используемого брокера сообщений. Например, следующее объявление `<bean>` определяет очередь ActiveMQ:

```
<bean id="queue" class="org.apache.activemq.command.ActiveMQQueue">
    <constructor-arg value="spitter.queue"/>
</bean>
```

Аналогично следующее объявление `<bean>` определяет тему ActiveMQ:

```
<bean id="topic" class="org.apache.activemq.command.ActiveMQTopic">
    <constructor-arg value="spitter.topic"/>
</bean>
```

В обоих случаях элемент `<constructor-arg>` определяет имя очереди или темы, которая будет использоваться брокером сообщений – `spitter.topic` в последнем примере.

Как и в случае с фабрикой соединений, в пространстве имен ActiveMQ предоставляется альтернативный способ объявления очередей и тем. Для определения очереди можно использовать элемент `<amq:queue>`:

```
<amq:queue id="queue" physicalName="spitter.queue" />
```

А для определения темы – элемент `<amq:topic>`:

```
<amq:topic id="topic" physicalName="spitter.topic" />
```



В обоих случаях атрибут `physicalName` определяет имя канала обмена сообщениями.

К настоящему моменту было показано, как объявлять основные компоненты для отправки и приема сообщений. Теперь можно приступить к непосредственной реализации операций. Для этого воспользуемся классом `JmsTemplate`, входящим в состав Spring и являющимся основой поддержки JMS в Spring. Но, чтобы оценить все преимущества использования класса `JmsTemplate`, сначала посмотрим, как реализовать обмен сообщениями без него.

13.3. Работа с шаблонами JMS в Spring

Механизм JMS дает разработчикам приложений на языке Java стандартный API для взаимодействия с брокерами сообщений, а также для отправки и приема сообщений. Кроме того, практически все существующие реализации брокеров сообщений поддерживают JMS. Поэтому нет причин изучать частные API обмена сообщениями для организации взаимодействий с брокерами.

Да, JMS предлагает универсальный интерфейс для работы с любыми брокерами, но за это удобство приходится платить. Отправить и принять сообщение с помощью JMS намного сложнее, чем просто шлепнуть печать на конверт. Как будет показано далее, JMS требует, чтобы вы еще и заправили почтовый грузовик (фигурально выражаясь).

13.3.1. Борьба с разбуханием JMS-кода

В разделе 6.3.1 было показано, насколько громоздким может быть код для работы с JDBC, необходимый для обработки установления соединений, выполнения запросов, получения результатов и обработки исключений. К сожалению, JMS API подвержен той же болезни, как можно заметить в листинге 13.1.

Листинг 13.1. Отправка сообщения без привлечения Spring

```
ConnectionFactory cf =
    new ActiveMQConnectionFactory("tcp://localhost:61616");

Connection conn = null;
Session session = null;
try {
    conn = cf.createConnection();
```

```
session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
Destination destination = new ActiveMQQueue("spitter.queue");
MessageProducer producer = session.createProducer(destination);
TextMessage message = session.createTextMessage();

message.setText("Hello world!");
producer.send(message); // Отправка сообщения
} catch (JMSEException e) {
    // обработать исключение?
} finally {
    try {
        if (session != null) {
            session.close();
        }
        if (conn != null) {
            conn.close();
        }
    } catch (JMSEException ex) {
    }
}
}
```

Ох уж этот шаблонный код! Как и в примере использования JDBC, потребовалось более 20 строк кода, только чтобы отправить простое сообщение «Hello world!». При этом к отправке сообщения прямое отношение имеют лишь несколько строк, остальные выполняют подготовительные операции.

Не лучше выглядит и реализация приема сообщения, как показано в листинге 13.2.

Листинг 13.2. Прием сообщения без привлечения Spring

```
ConnectionFactory cf =
    new ActiveMQConnectionFactory("tcp://localhost:61616");

Connection conn = null;
Session session = null;
try {
    conn = cf.createConnection();
    conn.start();
    session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
    Destination destination =
        new ActiveMQQueue("spitter.queue");
    MessageConsumer consumer = session.createConsumer(destination);
    Message message = consumer.receive();
    TextMessage textMessage = (TextMessage) message;
```

```
System.out.println("GOT A MESSAGE: " + textMessage.getText());
conn.start();
} catch (JMSEException e) {
    // обработать исключение?
} finally {
    try {
        if (session != null) {
            session.close();
        }
        if (conn != null) {
            conn.close();
        }
    } catch (JMSEException ex) {
    }
}
```

И снова, как в листинге 13.1, здесь используется масса шаблонного кода. Если провести построчное сравнение, можно обнаружить, что эти два листинга практически идентичны. И если посмотреть на тысячи других примеров использования JMS, можно с тем же успехом заметить их поразительное сходство. Где-то вместо фабрики соединений используется JNDI, а где-то вместо очередей используются темы, но все они следуют одному и тому же шаблону.

Как следствие при работе с JMS вы постоянно будете обнаруживать, что пишете один и тот же код снова и снова. Хуже того, вы будете обнаруживать, что повторяете программный код, написанный другими разработчиками.

В главе 6 уже было показано, как использование класса `JdbcTemplate` позволяет избавиться от шаблонного кода при работе с JDBC. Теперь пришло время посмотреть, как тот же эффект помогает получить класс `JmsTemplate` при работе с JMS.

13.3.2. Работа с шаблонами JMS

Класс `JmsTemplate` – это ответ фреймворка Spring на необходимость писать массу шаблонного кода для работы с JMS. Класс `JmsTemplate` берет на себя все хлопоты по созданию соединений, открытию сеансов и приему/передаче сообщений. Он позволяет разработчику сосредоточиться на конструировании сообщений для передачи или обработке принимаемых сообщений.

Более того, `JmsTemplate` может обрабатывать все эти неудобные исключения `JMSEException`. Если в процессе работы в классе `JmsTemplate`

будет возбуждено исключение `JMSException`, оно будет перехвачено, и повторно будет возбуждено неконтролируемое исключение, являющееся одним из подклассов класса `JmsException`.

В табл. 13.1 перечислены неконтролируемые исключения, наследники `JmsException`, реализованные в фреймворке Spring, и соответствующие им стандартные исключения, наследники `JMSException`.

Таблица 13.1. Класс `JmsTemplate` перехватывает стандартные исключения `JMSException` и в ответ возбуждает соответствующие им исключения `JmsException`

Spring (<code>org.springframework.jms.*</code>)	JMS (<code>javax.jms.*</code>)
<code>DestinationResolutionException</code>	Не имеет аналога в JMS. Возбуждается, когда фреймворк Spring не может определить имя приемника
<code>IllegalStateException</code>	<code>IllegalStateException</code>
<code>InvalidClientIDException</code>	<code>InvalidClientIDException</code>
<code>InvalidDestinationException</code>	<code>InvalidDestinationException</code>
<code>InvalidSelectorException</code>	<code>InvalidSelectorException</code>
<code>JmsSecurityException</code>	<code>JmsSecurityException</code>
<code>ListenerExecutionFailedException</code>	Не имеет аналога в JMS. Возбуждается фреймворком Spring, когда выполнение метода приема завершается неудачей
<code>MessageConversionException</code>	Не имеет аналога в JMS. Возбуждается фреймворком Spring, когда преобразование сообщения завершается неудачей
<code>MessageEOFException</code>	<code>MessageEOFException</code>
<code>MessageFormatException</code>	<code>MessageFormatException</code>
<code>MessageNotReadableException</code>	<code>MessageNotReadableException</code>
<code>MessageNotWritableException</code>	<code>MessageNotWritableException</code>
<code>ResourceAllocationException</code>	<code>ResourceAllocationException</code>
<code>SynchedLocalTransactionFailedException</code>	Не имеет аналога в JMS. Возбуждается фреймворком Spring, когда синхронизированная локальная транзакция не может быть завершена
<code>TransactionInProgressException</code>	<code>TransactionInProgressException</code>
<code>TransactionRolledBackException</code>	<code>TransactionRolledBackException</code>
<code>UncategorizedJmsException</code>	Не имеет аналога в JMS. Возбуждается фреймворком Spring в ситуациях, когда никакое другое исключение неприменимо

К чести JMS API, иерархия `JMSException` включает богатый набор подклассов, позволяющих определить причины, повлекшие исключи-

чительную ситуацию. Однако все подклассы `JMSEException` являются контролируемыми исключениями и потому обязательно должны перехватываться. Класс `JmsTemplate` автоматически перехватывает все эти исключения и возбуждает соответствующие им неконтролируемые исключения, наследующие класс `JMSEException`.

История о двух версиях класса `JmsTemplate`. Фактически в состав фреймворка Spring включены два класса шаблонов JMS: `JmsTemplate` и `JmsTemplate102`. `JmsTemplate102` – это специализированная версия класса `JmsTemplate`, предназначенная для взаимодействий с провайдерами JMS 1.0.2. В спецификации JMS 1.0.2 темы и очереди интерпретируются совершенно иначе – как домены. В спецификации JMS 1.1+ темы и очереди объединяются прикладным интерфейсом, не зависящим от доменов. Поскольку темы и очереди в JMS 1.0.2 интерпретируются иначе, для взаимодействия со старыми реализациями JMS был создан специализированный класс `JmsTemplate102`. В этой главе предполагается использование современного провайдера JMS, поэтому все свое внимание мы сосредоточим исключительно на классе `JmsTemplate`.

Внедрение шаблона JMS

Чтобы задействовать класс `JmsTemplate`, необходимо объявить соответствующий компонент в конфигурационном файле Spring, как показано ниже:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
</bean>
```

Поскольку экземпляру `JmsTemplate` нужно знать, как приобрести соединение с брокером сообщений, в свойстве `connectionFactory` компонента следует указать ссылку на компонент, реализующий интерфейс JMS `ConnectionFactory`. В примере выше внедряется ссылка на компонент `connectionFactory`, объявленный выше, в разделе 13.2.1.

Вот и все, что необходимо, чтобы задействовать класс `JmsTemplate`, – теперь все готово к работе. Начнем с отправки сообщений!

Отправка сообщений

Одной из особенностей, которую предполагается встроить в приложение Spitter, является возможность извещения (возможно, по электронной почте) других пользователей о создании нового сообщения. Эту функцию можно было бы добавить непосредственно в том месте, где производится создание сообщения. Но фактическая

отправка извещений в этом месте может занять продолжительное время и отрицательно сказаться на субъективном восприятии производительности приложения. При добавлении нового сообщения приложение должно откликаться как можно быстрее.

Чтобы не тратить времени на отправку извещений во время добавления сообщения, проще будет поместить это извещение в очередь и выполнить его рассылку позднее, после отправки ответа пользователю. Время, необходимое на передачу извещения в очередь или тему, весьма незначительно, особенно если сравнить со временем, необходимым для рассылки извещений другим пользователям.

Для поддержки асинхронной рассылки извещений введем в приложение Spittle службу `AlertService`:

```
package com.habuma.spitter.alerts;
import com.habuma.spitter.domain.Spittle;
public interface AlertService {
    void sendSpittleAlert(Spittle spittle);
}
```

Как видите, `AlertService` – это интерфейс, определяющий единственный метод `sendSpittleAlert()`. Класс `AlertServiceImpl` – это реализация интерфейса `AlertService`, использующая компонент `JmsTemplate` для отправки объектов `Spittle` в очередь сообщений с целью их обработки в более позднее время.

Листинг 13.3. Отправка сообщения с помощью `JmsTemplate`

```
package com.habuma.spitter.alerts;

import javax.jms.JMSEException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

import com.habuma.spitter.domain.Spittle;

public class AlertServiceImpl implements AlertService {
    public void sendSpittleAlert(final Spittle spittle) {
        jmsTemplate.send(                                // Отправка сообщения
            "spittle.alert.queue",                      // Имя приемника
            new MessageCreator() {
                public Message createMessage(Session session) {
                    // Создание сообщения
                }
            }
        );
    }
}
```

```

        new MessageCreator() {
            public Message createMessage(Session session)
                throws JMSException {
                return session.createObjectMessage(spittle); // Создание
            } // сообщения
        );
    }

    @Autowired
    JmsTemplate jmsTemplate;
}

```

Первый параметр метода `send()` класса `JmsTemplate` – имя приемника JMS, куда отправляется сообщение. При вызове метода `send()` класс `JmsTemplate` приобретет JMS-соединение и сеанс и отправит сообщение от имени отправителя (рис. 13.5).

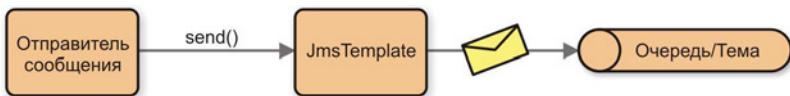


Рис. 13.5. Класс `JmsTemplate` берет на себя все сложности, связанные с отправкой сообщений

Что касается самого сообщения, оно конструируется с использованием реализации интерфейса `MessageCreator`, организованной в виде анонимного вложенного класса. В методе `createMessage()` реализации интерфейса `MessageCreator` программа просто предлагает сеансу создать новый объект сообщения и добавить в него объект `Spittle`.

Вот и все! Обратите внимание, что метод `sendSpittleAlert()` занимается исключительно сборкой и отправкой сообщения. В нем отсутствует код, управляющий соединением или сеансом, – все это автоматически делает класс `JmsTemplate`. И в нем отсутствует код, перехватывающий исключения `JMSException`, – класс `JmsTemplate` перехватывает все исключения `JMSException` и возбуждает в ответ неконтролируемы исключения, перечисленные в табл. 13.1.

Настройка приемника по умолчанию

В листинге 13.3 явно был указан конкретный приемник, куда будут отправляться сообщения методом `send()`. Такую форму метода `send()` удобно использовать, когда требуется организовать выбор

приемника программно. Но в реализации `AlertServiceImpl` все сообщения отправляются в один и тот же приемник, поэтому преимущества данной формы метода `send()` неочевидны.

Вместо явного определения приемника при отправке каждого сообщения в объявлении компонента `JmsTemplate` можно было бы определить приемник по умолчанию:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="defaultDestinationName"
              value="spittle.alert.queue"/>
</bean>
```

Теперь вызов метода `send()` класса `JmsTemplate` можно немного упростить, убрав первый параметр:

```
jmsTemplate.send(
    new MessageCreator() {
        ...
    }
);
```

Эта форма метода `send()` принимает только реализацию интерфейса `MessageCreator`, а в качестве приемника используется приемник по умолчанию.

Извлечение сообщений

Выше было показано, как с помощью `JmsTemplate` отправлять сообщения. А как быть на принимающей стороне? Можно ли использовать класс `JmsTemplate` для приема сообщений?

Да, можно. Фактически прием сообщений с помощью класса `JmsTemplate` реализуется еще проще. Чтобы извлечь сообщение, достаточно вызвать метод `receive()` класса `JmsTemplate`, как показано в листинге 13.4.

Листинг 13.4. Прием сообщения с помощью класса `JmsTemplate`

```
public Spittle getAlert() {
    try {
        ObjectMessage receivedMessage =
            (ObjectMessage) jmsTemplate.receive(); // Прием сообщения
```

```

        return (Spittle) receivedMessage.getObject(); // Извлечение объекта
    } catch (JMSException jmsException) {
        // Возбудить преобразованное исключение
        throw JmsUtils.convertJmsAccessException(jmsException);
    }
}

```

Метод `receive()` класса `JmsTemplate` пытается получить сообщение, обратившись к брокеру сообщений. Если сообщение недоступно, метод `receive()` будет ждать, пока сообщение не станет доступно. Это взаимодействие изображено на рис. 13.6.

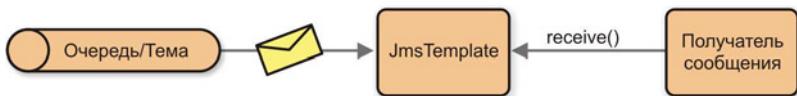


Рис. 13.6. Прием сообщений из темы или из очереди с помощью класса `JmsTemplate` осуществляется простым вызовом его метода `receive()`. Обо всем остальном позаботится сам класс `JmsTemplate`

Поскольку известно, что в приложении Spitter отправляемые сообщения содержат объект, их можно привести к типу `ObjectMessage` при приеме. А затем вызвать метод `getObject()`, чтобы извлечь объект `Spittle` из сообщения `ObjectMessage` и вернуть его.

Единственная неприятность, которая здесь поджидает, – необходимость предпринять какие-то действия в случае появления исключения `JMSException`. Как уже упоминалось, класс `JmsTemplate` прекрасно справляется со всеми контролируемыми исключениями, наследниками `JMSException`, возбуждая в ответ соответствующие им неконтролируемые исключения, наследники `JmsException`. Но это относится только к вызовам методов класса `JmsTemplate`. Класс `JmsTemplate` не сможет помочь обработать исключение `JMSException`, которое может быть возбуждено методом `getObject()` класса `ObjectMessage`.

Поэтому необходимо либо перехватить исключение `JMSException`, либо объявить, что данный метод может возбуждать его. В соответствии с философией фреймворка Spring, которая призывает избегать контролируемых исключений, было решено не позволять исключению `JMSException` покидать пределы этого метода и перехватить его. В блоке `catch` можно воспользоваться методом `convertJmsAccessException()` класса `JmsUtils`, входящего в состав фреймворка Spring, чтобы преобразовать контролируемое исключение `JMSException` в не-

контролируемое исключение `JmsException`. В действительности именно так класс `JmsTemplate` преобразует исключения.

Самым большим недостатком операции извлечения сообщений с помощью класса `JmsTemplate` является синхронность метода `receive()`. Это означает, что приложение вынуждено будет ждать появления сообщения, так как метод `receive()` окажется заблокированным до момента, пока сообщение не станет доступно (или пока не истечет предельное время ожидания). Не кажется ли вам странной синхронность при приеме сообщения, которое посыпается асинхронно?

Решить эту проблему нам помогут простые Java-объекты (POJO), управляемые сообщениями. Посмотрим, как организовать асинхронный прием сообщений с помощью компонентов, реагирующих на сообщения, а не ожидающих их появления.

13.4. Создание POJO, управляемых сообщениями

Во время учебы в колледже в одни из летних каникул мне довелось работать в Йеллоустонском национальном парке (Yellowstone National Park). Это была не такая высококвалифицированная работа, как у смотрителя парка или у парня, включающего и выключающего гейзер «Старый служака» (Old Faithful). Я работал в гостинице «Old Faithful Inn» – менял белье, чистил ванные и пылесосил полы. Не очень интересно, зато я работал в одном из красивейших мест на Земле.

Каждый день после работы я заходил на почту, чтобы узнать, пришла ли мне какая-то почта. Я отсутствовал дома несколько недель, и было очень приятно получить письмо или открытку от моих школьных друзей. У меня не было своего почтового ящика, поэтому мне нужно было приходить на почту самому и спрашивать у человека, сидящего на табурете за стойкой, не пришло ли мне что-нибудь. С этого момента начиналось ожидание.

Человеку за стойкой было примерно 195 лет. И как всякому в этом возрасте, ему было тяжело ходить. Он отрывал свой зад от табурета, медленно срывался с места и исчезал за загородкой. Спустя некоторое время он появлялся, возвращался к стойке и водружал себя на табурет. Затем оглядывал меня и говорил: «Сегодня – ничего».

Метод `receive()` класса `JmsTemplate` действует практически так же, как тот служащий на почте. Когда программа вызывает метод `receive()`, он выходит и проверяет наличие сообщения в очереди или



в теме, но не возвращается, пока сообщение не поступит или пока не истечет предельное время ожидания. В это время приложение «стоит возле стойки» и ничего не делает. Разве не было бы лучше, если бы приложение могло продолжать заниматься своими делами и извещалось бы в момент поступления сообщения?

Одним из отличительных моментов спецификации EJB 2 было включение в нее компонентов, управляемых сообщениями (Message-Driven Beans, MDB). Компоненты MDB – это компоненты EJB, обрабатывающие сообщения асинхронно. Иными словами, MDB реагируют на сообщения, появляющиеся в приемнике JMS, как на события. Этим они в корне отличаются от компонентов, выполняющих прием сообщений синхронно, выполнение которых блокирует-ся в отсутствие сообщений.

Компоненты MDB ярко выделялись на общем ландшафте EJB. Даже самые непримиримые критики EJB признавали, что компоненты MDB являются весьма элегантным способом обработки сообщений. Единственным недостатком компонентов MDB в спецификации EJB 2 была необходимость поддержки ими интерфейса javax.ejb.MessageDrivenBean. При этом они также должны были реализовать некоторые методы обратного вызова поддержки жизненного цикла. Проще говоря, компоненты MDB были далеко не простыми Java-объектами.

Компоненты MDB были убраны из спецификации EJB 3, где предпочтение было отдано более простым POJO. Теперь больше не требуется включать реализацию интерфейса MessageDrivenBean. Вместо этого требуется реализовать более обобщенный интерфейс javax.jms.MessageListener и использовать аннотацию @MessageDriven.

Начиная с версии 2.0, фреймворк Spring решил проблему асинхронного чтения сообщений, предоставив собственную реализацию компонентов, управляемых сообщениями, которые очень похожи на компоненты MDB, определяемые спецификацией EJB 3. В этом разделе рассказывается о поддержке асинхронного чтения сообщений в Spring с применением POJO, управляемых сообщениями (назовем их MDP для краткости).

13.4.1. Создание объекта для чтения сообщений

Если бы потребовалось реализовать прием извещений о новых сообщениях в приложении Spitter с использованием модели, определяемой спецификацией EJB, нам пришлось бы задействовать анно-

тацию @MessageDriven. И реализовать интерфейс MessageListener, хотя это и не обязательно. Результат выглядел бы, как показано ниже:

```
@MessageDriven(mappedName="jms/spittle.alert.queue")
public class SpittleAlertHandler implements MessageListener {
    @Resource
    private MessageDrivenContext mdc;

    public void onMessage(Message message) {
        ...
    }
}
```

На мгновение представьте более простой мир, где от компонентов, управляемых сообщениями, не требуется реализовать интерфейс MessageListener. В этом счастливом мире небо всегда было бы голубым, птицы всегда насвистывали бы вашу любимую песню, и вам не пришлось бы писать реализацию метода onMessage() или внедрять компонент MessageDrivenContext.

Ладно, требования спецификации EJB 3, предъявляемые к MDB, возможно, не самые сложные. Но сам факт, что реализация SpittleAlertHandler оказывается слишком тесно связанный с API спецификации EJB и далека от POJO, весьма неприятен. В идеале хотелось бы, чтобы обработчик извещений был способен обрабатывать сообщения, но был реализован так, чтобы не иметь тесной зависимости от конкретного API.

Фреймворк Spring обеспечивает возможность задействовать метод простого объекта POJO для обработки сообщений, извлекаемых из очереди или из темы JMS. Например, в листинге 13.5 представлена POJO-реализация класса SpittleAlertHandler, которой вполне достаточно для нужд приложения.

Листинг 13.5. Spring MDP, асинхронно получающий и обрабатывающий сообщения

```
package com.habuma.spitter.alerts;
import com.habuma.spitter.domain.Spittle;

public class SpittleAlertHandler {

    public void processSpittle(Spittle spittle) {          // Метод-обработчик
        // ...здесь находится реализация метода...
    }
}
```

Хотя управление цветом неба и дрессировка птиц вне компетенции фреймворка Spring, тем не менее листинг 13.5 демонстрирует, что сказочный мир, описанный выше, не так уж и далек от реальности. Мы еще вернемся к реализации метода `processSpittle()`. А пока обратите внимание, что в классе `SpittleAlertHandler` нет ничего, что говорило бы о связи с JMS. Это самый простой, самый обычный объект POJO во всех смыслах этого слова. Однако он может обрабатывать сообщения, подобно своим собратьям, компонентам EJB. Все, что ему требуется для работы, – это некоторые специальные настройки в конфигурационном файле Spring.

13.4.2. Настройка обработчиков сообщений

Вся хитрость наделения объекта POJO возможностью получать сообщения заключается в настройке его как обработчика. Пространство имен `jms` в Spring содержит все необходимое для этого. Сначала обработчик необходимо объявить компонентом:

```
<bean id="spittleHandler"
      class="com.habuma.spitter.alerts.SpittleAlertHandler" />
```

Затем, чтобы превратить `SpittleAlertHandler` в объект POJO, управляемый сообщениями, этот компонент можно объявить *обработчиком сообщений*:

```
<jms:listener-container connection-factory="connectionFactory">
    <jms:listener destination="spitter.alert.queue"
        ref="spittleHandler" method="processSpittle" />
</jms:listener-container>
```

Здесь, внутри контейнера обработчиков сообщений, объявляется обработчик сообщений. *Контейнер обработчиков сообщений* – это специальный компонент, просматривающий приемник JMS в ожидании поступления сообщений. Как только появится новое сообщение, он тут же извлечет его и передаст любому обработчику сообщений, заинтересованному в этом сообщении. Этот порядок взаимодействий изображен на рис. 13.7.

Для настройки контейнеров и обработчиков в Spring используется два элемента из пространства имен `jms`. Элемент `<jms:listener-container>` применяется для включения элементов `<jms:listener>`.



Рис. 13.7. Контейнер обработчика сообщений в очереди/теме.
При появлении сообщения он передает его обработчику
(такому как MDP)

В этом примере в его атрибуте connectionFactory указывается ссылка на компонент connectionFactory, который будет использоваться всеми вложенными элементами <jms:listener> при приеме сообщений. В данном случае атрибут connectionFactory можно было бы опустить, поскольку здесь он получает значение connectionFactory, предусмотренное по умолчанию.

Элемент <jms:listener> используется для идентификации компонента и его метода, который будет вызываться для обработки входящих сообщений. Чтобы обеспечить обработку извещений о новых сообщениях, в атрибут ref элемента записана ссылка на наш компонент spittleHandler. Когда в очереди spitter.alert.queue (определяется атрибутом destination) появится новое сообщение, будет вызван метод processSpittle() компонента spittleHandler (согласно атрибуту method).

13.5. Механизмы RPC, основанные на сообщениях

В главе 11 рассказывалось о нескольких механизмах, поддерживаемых фреймворком Spring, экспортования методов компонентов как удаленных служб и вызовов этих служб со стороны клиентов. В этой главе было показано, как реализовать обмен сообщениями между приложениями посредством очередей и тем JMS. Теперь попробуем объединить эти две концепции и посмотрим, как реализовать вызов удаленных методов, используя JMS в качестве транспорта.

Фреймворк Spring поддерживает два механизма RPC, основанных на сообщениях:

- ❑ сам фреймворк Spring предлагает компонент JmsInvokerServiceExporter, экспортирующий другие компоненты как службы, основанные на сообщениях, и компонент JmsInvokerProxyFactoryBean – для клиентов, пользующихся этими службами;

- проект Lingo предлагает аналогичный подход для реализации удаленных взаимодействий на основе сообщений с применением его компонентов `JmsServiceExporter` и `JmsProxyFactoryBean`.

Как будет показано ниже, эти два механизма очень похожи друг на друга, но каждый из них имеет свои достоинства и недостатки. Я покажу вам оба подхода, а какой из них лучше подходит для вас, вы решите сами. Начнем с первого механизма реализации JMS-служб, включенного в состав фреймворка Spring.

13.5.1. Механизм RPC, основанный на сообщениях, в фреймворке Spring

Как рассказывалось в главе 11, фреймворк Spring предоставляет несколько способов экспортования компонентов в виде удаленных служб. Для экспортования компонентов в виде RMI-служб мы использовали класс `RmiServiceExporter`, в виде служб Hessian и Burlap – классы `HessianExporter` и `BurlapExporter`, в виде служб HTTP Invoker – класс `HttpInvokerServiceExporter`. Но в Spring имеется еще один экспортер служб, о котором не рассказывалось в главе 11.

Экспортование JMS-службы

Класс `JmsInvokerServiceExporter` близко напоминает другие классы-экспортеры служб. Обратите внимание на некоторое сходство имен `JmsInvokerServiceExporter` и `HttpInvokerServiceExporter`. Если известно, что класс `HttpInvokerServiceExporter` экспортитрует службы, действующие по протоколу HTTP, то класс `JmsInvokerServiceExporter` должен экспортитровать службы, действующие по протоколу JMS.

Чтобы показать, как действует класс `JmsInvokerServiceExporter`, обратимся к реализации класса `AlertServiceImpl`, представленной в листинге 13.6.

Листинг 13.6. AlertServiceImpl – это POJO, обрабатывающий сообщения JMS

```
package com.habuma.spitter.alerts;

import org.springframework.mail.SimpleMailMessage;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.stereotype.Component;

import com.habuma.spitter.domain.Spittle;
```

```
@Component("alertService")
public class AlertServiceImpl implements AlertService {
    private JavaMailSender mailSender;
    private String alertEmailAddress;
    public AlertServiceImpl(JavaMailSender mailSender,
                           String alertEmailAddress) {
        this.mailSender = mailSender;
        this.alertEmailAddress = alertEmailAddress;
    }

    // Отправляет сообщение
    public void sendSpittleAlert(final Spittle spittle) {
        SimpleMailMessage message = new SimpleMailMessage();
        String spitterName = spittle.getSpitter().getFullName();
        message.setFrom("noreply@spitter.com");
        message.setTo(alertEmailAddress);
        message.setSubject("New spittle from " + spitterName);
        message.setText(spitterName + " says: " + spittle.getText());
        mailSender.send(message);
    }
}
```

Пока не пытайтесь разобраться во всех тонкостях метода `sendSpittleAlert()`. Подробнее о том, как отправлять почту с помощью Spring, будет рассказываться ниже, в разделе 15.3. Самое важное, на что следует обратить внимание: `AlertServiceImpl` – это простой объект POJO и не содержит ничего, что указывало бы на его использование для обработки сообщений JMS. Он реализует простой интерфейс `AlertService`, представленный ниже:

```
package com.habuma.spitter.alerts;
import com.habuma.spitter.domain.Spittle;

public interface AlertService {
    void sendSpittleAlert(Spittle spittle);
}
```

Как видите, `AlertServiceImpl` отмечен аннотацией `@Component`, благодаря которой он обнаруживается и регистрируется как компонент в контексте приложения Spring с идентификатором `alertService`. Ссылка на этот компонент будет использоваться при настройке компонента `JmsInvokerServiceExporter`:

```
<bean id="alertServiceExporter"
      class="org.springframework.jms.remoting.JmsInvokerServiceExporter"
      p:service-ref="alertService"
      p:serviceInterface="com.habuma.spitter.alerts.AlertService" />
```

Свойства этого компонента описывают, как должна выглядеть экспортируемая служба. В свойство service внедряется ссылка на компонент alertService, являющийся реализацией удаленной службы. А в свойстве serviceInterface указывается полное имя класса интерфейса, предоставляемого службой.

Свойства экспортара ничего не говорят о том, что служба будет доступна по протоколу JMS. Все дело в том, что JmsInvokerServiceExporter определяется как обработчик JMS. То есть его можно объявить внутри элемента `<jms:listener-container>`:

```
<jms:listener-container connection-factory="connectionFactory">
    <jms:listener destination="spitter.alert.queue"
                   ref="alertServiceExporter" />
</jms:listener-container>
```

Контейнеру обработчиков JMS передается ссылка на фабрику соединений, благодаря чему он получает возможность установить соединение с брокером сообщений. А в элементе `<jms:listener>` указывается имя приемника JMS, куда будут доставляться сообщения.

Доступ к JMS-службе

На данный момент JMS-служба извещений готова к работе и ожидает появления RPC-сообщений в очереди с именем `spitter.alert.queue`. Для доступа к этой службе со стороны клиента будет использоваться компонент `JmsInvokerProxyFactoryBean`.

Компонент `JmsInvokerProxyFactoryBean` очень близко напоминает другие фабричные компоненты, представленные в главе 11, производящие прокси-объекты для доступа к удаленным службам. Он скрывает детали взаимодействий с удаленной службой за удобным интерфейсом, посредством которого клиент взаимодействует со службой. Основное отличие заключается в том, что `JmsInvokerProxyFactoryBean` создает прокси-объекты не для доступа к службам, действующим по протоколу RMI или HTTP, а для доступа к службам, действующим по протоколу JMS и экспортируемым компонентом `JmsInvokerServiceExporter`.

Для доступа к службе извещений компонент `JmsInvokerProxyFactoryBean` можно настроить, как показано ниже:

```
<bean id="alertService"
      class="org.springframework.jms.remoting.JmsInvokerProxyFactoryBean">
    <property name="connectionFactory" ref="connectionFactory" />
    <property name="queueName" value="spitter.alert.queue" />
    <property name="serviceInterface"
              value="com.habuma.spitter.alerts.AlertService" />
</bean>
```

Свойства `connectionFactory` и `queueName` определяют, как должны доставляться RPC-сообщения. В данном случае – брокеру сообщений, указанному в настройках фабрики соединений, в очередь с именем `spitter.alert.queue`. Свойство `serviceInterface` указывает, что прокси-объект должен действовать через интерфейс `AlertService`.

Компоненты `JmsInvokerServiceExporter` и `JmsInvokerProxyFactoryBean` реализуют в Spring альтернативный способ организации удаленных взаимодействий по протоколу JMS. Но они – не единственный способ экспортирования и использования служб на основе JMS. И, возможно, они являются даже не самым оптимальным выбором. Познакомимся с особенностями проекта Lingo и посмотрим, что он может предложить сверх того, что предлагает механизм JMS Invoker.

13.5.2. Механизм RPC, основанный на сообщениях, в Lingo

Lingo¹ – это механизм удаленных взаимодействий, основанный на фреймворке Spring и напоминающий механизм JMS Invoker в Spring. Фактически в документации Javadoc с описанием классов JMS Invoker библиотека Lingo упоминается косвенно, как образец для подражания².

В отличие от JMS Invoker, библиотека Lingo способна в полной мере использовать асинхронную природу JMS для обращения к службам. Это означает, что на момент, когда клиент производит вызов, сервер вообще может быть недоступным. Более того, при взаимодействии со службой, выполняющей длительные операции, клиент может не ждать их завершения.

¹ <http://lingo.codehaus.org>.

² Библиотека Lingo не упоминается напрямую, зато упоминается ее автор, Джеймс Страхан (James Strachan).

В отличие от других механизмов удаленных взаимодействий, обсуждавшихся в главе 11, и от механизма JMS Invoker, библиотека Lingo не входит в состав фреймворка Spring Framework. Это отдельный проект, опирающийся на использование экспортера JMS-служб и клиентского прокси-объекта.

Наше знакомство с библиотекой Lingo мы начнем с обзора библиотечного класса `JmsServiceExporter`, используемого для экспортации служб. Затем мы рассмотрим приемы доступа к этим службам с помощью библиотечного класса `JmsServiceProxy`.

Экспортирование асинхронной службы

Как показано в следующем объявлении, компоненты `JmsServiceExporter` и `JmsInvokerServiceExporter` настраиваются практически одинаково:

```
<bean id="alertServiceExporter"
      class="org.logicblaze.lingo.jms.JmsServiceExporter"
      p:connectionFactory-ref="connectionFactory"
      p:destination-ref="alertServiceQueue"
      p:service-ref="alertService"
      p:serviceInterface="com.habuma.spitter.alerts.AlertService" />
```

Свойства `service` и `serviceInterface` в точности соответствуют одноименным свойствам компонента `JmsInvokerServiceExporter`. Однако компонент `JmsServiceExporter` имеет новое свойство. Компонент `JmsServiceExporter` не может использоваться как обработчик сообщений в контейнере обработчиков, поэтому ему нужно передать ссылки на фабрику соединений JMS и приемник сообщений в свойствах `connectionFactory` и `destination`.

Обратите внимание, что свойство `destination` имеет тип `javax.jms.Destination`. По этой причине в него должна внедряться ссылка на компонент-приемник. Следующий компонент `alertServiceQueue` будет играть роль приемника JMS RPC-сообщений, передаваемых в очередь с именем `spittle.alert.queue`:

```
<amq:queue id="alertServiceQueue"
            physicalName="spittle.alert.queue" />
```

На данный момент не было выявлено ничего, что дает библиотека Lingo сверх возможностей механизма JMS Invoker. Поэтому у кого-то может возникнуть вопрос, зачем я рассказываю здесь об этой

библиотеке, если фреймворк Spring имеет собственный механизм JMS RPC, предоставляющий те же самые возможности.

Но не спешите с выводами. Как демонстрируется чуть ниже, на стороне клиента библиотека Lingo предлагает кое-что, чего не может предложить JMS Invoker: асинхронные обращения к службе.

Доступ к асинхронной службе

При вызове методов прокси-объекта `JmsInvokerServiceProxy`, созданного фреймворком Spring, приложение вынуждено ждать. Даже несмотря на то что в качестве транспорта используется JMS, прокси-объект будет ждать получения ответа от службы.

Компонент `JmsProxyFactoryBean`, создаваемый библиотекой Lingo, напротив, может быть настроен на выполнение операций в асинхронном режиме. Например, клиент службы извещений, реализованный на основе библиотеки Lingo, можно настроить, как показано ниже:

```
<bean id="alertService"
    class="org.logicblaze.lingo.jms.JmsProxyFactoryBean"
    p:connectionFactory-ref="connectionFactory"
    p:destination-ref="queue"
    p:serviceInterface="com.habuma.spitter.alerts.AlertService">
    <property name="metadataStrategy">
        <bean id="metadataStrategy"
            class="org.logicblaze.lingo.SimpleMetadataStrategy">
            <constructor-arg value="true"/>
        </bean>
    </property>
</bean>
```

Свойства `connectionFactory`, `destination` и `serviceInterface` играют ту же роль, что и в предыдущих примерах. Новым здесь является свойство `metadataStrategy`, значением которого является внутренний компонент типа `SimpleMetadataStrategy`.

Кроме всего прочего, свойство `metadataStrategy` определяет, какие методы должны действовать в асинхронном режиме. Единственной доступной реализацией является класс `SimpleMetadataStrategy`, конструктор которого может принимать единственный логический аргумент, определяющий, должны ли все методы, не имеющие возвращаемого значения, действовать асинхронно. В данном случае конструктору передается значение `true`, указывающее, что все методы



обращения к службе, не имеющие возвращаемого значения, должны действовать асинхронно и возвращать управление немедленно.

Если передать конструктору значение `false` или вообще исключить определение свойства `metadataStrategy` из объявления компонента `JmsProxyFactoryBean`, все методы обращения к службе будут действовать синхронно, а компонент `JmsProxyFactoryBean` превратится практически в полный аналог компонента `JmsInvokerServiceProxy`.

13.6. В заключение

Асинхронные механизмы обмена сообщениями имеют ряд преимуществ перед синхронными механизмами RPC. Косвенные взаимодействия ослабляют связь между приложениями и тем самым снижают влияние перерывов в обслуживании на работу клиентских приложений. Кроме того, благодаря тому что сообщения доставляются получателям с помощью посредника, отправителю не приходится ждать ответа. Во многих случаях это может существенно повысить производительность приложения.

Несмотря на то что JMS предоставляет стандартный API, доступный для любых Java-приложений, желающих участвовать в асинхронных взаимодействиях, его использование может оказаться весьма утомительным занятием. Фреймворк Spring устраниет необходимость писать шаблонный код, необходимый для работы с механизмом JMS, и существенно упрощает реализацию асинхронного обмена сообщениями.

В этой главе мы познакомились с несколькими способами организации асинхронных взаимодействий между двумя приложениями, осуществляемых с помощью Spring посредством брокеров сообщений и JMS. Шаблон JMS в Spring позволяет избежать шаблонного кода, который обычно приходится писать при использовании традиционной программной модели JMS. А управляемые сообщениями компоненты в Spring дают возможность объявлять методы, вызываемые автоматически при появлении новых сообщений в очереди или в теме.

Кроме того, здесь был рассмотрен пример применения механизма JMS Invoker, реализованного в Spring, а также библиотеки Lingo, обеспечивающих RPC на основе сообщений с помощью компонентов Spring. Несмотря на то что механизм JMS Invoker в Spring был создан по образу и подобию библиотеки Lingo и может служить ее заменой, он поддерживает только синхронные взаимодействия. Биб-

библиотека Lingo, в свою очередь, предлагает дополнительные особенности, недоступные в JMS Invoker: возможность вызывать удаленные методы асинхронно.

Теперь, когда мы знаем, как Spring позволяет упростить использование JMS, посмотрим, как фреймворк Spring помогает использовать стандартный Java-механизм с похожим названием. В следующей главе мы исследуем возможность экспортования компонентов Spring как управляемых компонентов, с помощью JMX.



Глава 14. Управление компонентами Spring с помощью JMX

В этой главе рассматриваются следующие темы:

- ❑ экспорттирование компонентов Spring как управляемых компонентов;
- ❑ удаленное управление компонентами Spring;
- ❑ обработка извещений JMX.

Поддержка DI в Spring – отличный способ настройки значений свойств компонентов приложения. Но после развертывания и запуска приложения механизм будет DI не в состоянии оказать помощь в изменении конфигурации. Представьте, что вам потребовалось вторгнуться в действующее приложение и изменить настройки на лету. В этой ситуации на выручку может прийти механизм *Java Management Extensions* (JMX).

JMX – это технология, позволяющая снабдить приложение средствами управления, мониторинга и настройки. Первоначально доступная как отдельное расширение языка Java, в настоящее время технология JMX является стандартной частью Java 5.

Ключевым компонентом приложения, снабженного средствами управления с помощью JMX, является *управляемый компонент* (Managed Bean, MBean). MBean – это компонент JavaBean, экспорттирующий некоторые методы, которые определяют интерфейс управления. Спецификация JMX определяет четыре типа компонентов MBean.

- ❑ *Стандартные компоненты MBean* – это такие компоненты MBean, интерфейсы управления которыми определяются фиксированными Java-интерфейсами, реализуемыми классом компонента.
- ❑ *Динамические компоненты MBean* – это такие компоненты MBean, интерфейсы управления которыми определяются во время выполнения вызовами методов интерфейса DynamicMBean.

Поскольку интерфейс управления определяется не статическими интерфейсами, он может изменяться во время выполнения.

- ❑ *Открытые компоненты MBean* – особая разновидность динамических компонентов MBean, чьи атрибуты и операции ограничиваются простыми типами, классами-обертками вокруг простых типов и любыми типами, которые могут быть разложены на простые типы или классы-обертки вокруг простых типов.
- ❑ *Компоненты-модели MBean* – особая разновидность динамических компонентов MBean, соединяющих в себе интерфейс управления и ресурс, управление которым он осуществляет. Компоненты-модели не столько пишутся, сколько объявляются. Обычно они создаются с помощью фабрики, которая использует некоторую метаинформацию, необходимую для сборки интерфейса управления.

Модуль JMX в Spring позволяет экспортить компоненты Spring как управляемые компоненты-модели, что дает возможность заглянуть внутрь приложения и изменить его конфигурацию даже во время его выполнения. Посмотрим, как снабдить наше приложение Spitter поддержкой JMX, чтобы можно было управлять компонентами в контексте приложения Spring.

14.1. Экспортирование компонентов Spring как управляемых компонентов

Существует несколько способов использования JMX для управления компонентами внутри приложения Spitter. Для простоты начнем с самого скромного изменения – добавим в контроллер HomeController новое свойство spittlesPerPage:

```
public static final int DEFAULT_SPITTLES_PER_PAGE = 25;

private int spittlesPerPage = DEFAULT_SPITTLES_PER_PAGE;

public void setSpittlesPerPage(int spittlesPerPage) {
    this.spittlesPerPage = spittlesPerPage;
}

public int getSpittlesPerPage() {
    return spittlesPerPage;
}
```

Прежде, когда HomeController вызывал метод getRecentSpittles() класса SpitterService, он передавал ему константу DEFAULT_SPITTLIES_PER_PAGE, обеспечивая вывод не более 25 сообщений на главной странице. Теперь же решение о количестве сообщений будет приниматься не только на этапе сборки приложения, но и может быть изменено во время выполнения, с помощью механизма JMX. Новое свойство spittlesPerPage – это первый шаг к намеченной цели.

Но само по себе свойство spittlesPerPage не поддерживает возможности изменения извне количества сообщений, отображаемых на главной странице. Это всего лишь свойство компонента, подобное любому другому свойству. Поэтому необходимо сделать следующий шаг и экспортить компонент HomeController как управляемый компонент MBean. В результате этого шага свойство spittlesPerPage будет экспортировано как *управляемый атрибут* компонента MBean и доступно для изменения.

Ключом к JMX-фикиации компонентов в Spring является MBeanExporter – компонент, экспортирующий один или более компонентов Spring как компоненты-модели (Model MBean) в сервер MBean. Сервер MBean (иногда называется *агентом MBean*) – это контейнер, где располагаются управляемые компоненты MBean и посредством которого осуществляется доступ к этим управляемым компонентам MBean.

Как показано на рис. 14.1, экспортование компонентов Spring в виде управляемых компонентов MBean позволяет вторгаться в выполняющееся приложение, просматривать свойства компонентов и вызывать их методы с помощью инструментов управления на основе JMX, таких как JConsole и VisualVM.

Следующий элемент <bean> объявляет компонент MBeanExporter в Spring, экспортующий компонент homeController как Model MBean:

```
<bean id="mbeanExporter"
      class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="spitter:name=HomeController"
              value-ref="homeController" />
      </map>
    </property>
</bean>
```

В этой простейшей форме компонент MBeanExporter можно настраивать посредством его свойства beans, внедряя в него отобра-

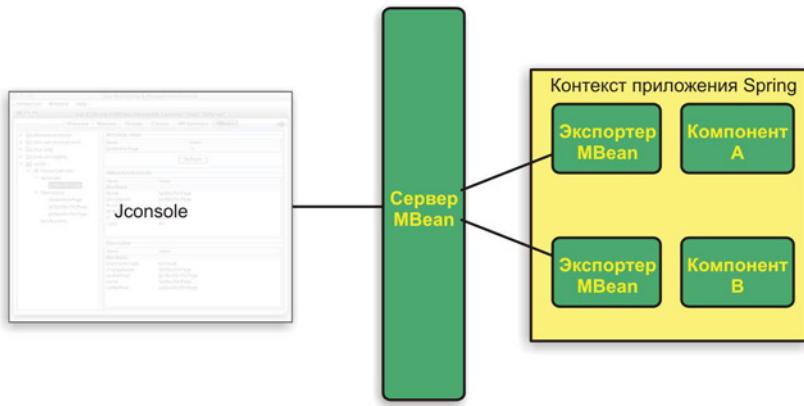


Рис. 14.1. Компонент Spring MBeanExporter экспортирует свойства и методы других компонентов Spring как атрибуты и операции JMX-сервера MBean. Благодаря этому появляется возможность заглянуть внутрь выполняющегося приложения с помощью таких JMX-инструментов управления, как JConsole

жение (`Map`) с описанием одного или более компонентов, которые требуется экспортить в виде управляемых компонентов-моделей MBean. Атрибут `key` в каждом элементе `<entry>` определяет имя, которое будет дано экспортируемому компоненту MBean (конструируется из имени домена управления и пары ключ/значение – `spitter:name=HomeController` в случае с управляемым компонентом `HomeController`). В атрибуте `value` элемента `<entry>` указывается ссылка на экспортируемый компонент Spring. Здесь экспортируется компонент `homeController`, что дает возможность управлять его свойствами во время выполнения посредством механизма JMX.

Откуда берется сервер MBean? Согласно настройкам, компонент MBeanExporter предполагает, что он выполняется внутри сервера приложений (такого как Tomcat) или в каком-то другом контексте, предоставляющем сервер MBean. Но если предполагается, что приложение на основе фреймворка Spring будет выполняться автономно или в контейнере, где отсутствует сервер MBean, необходимо настроить сервер MBean в контексте Spring. Эти настройки можно выполнить с помощью элемента `<context:mbean-server>`:

```
<context:mbean-server />
```

Элемент `<context:mbean-server>` создаст компонент, играющий роль сервера MBean в контексте приложения Spring. По умолчанию этот ком-

понент получит идентификатор `mbeanServer`. Учитывая это, его можно внедрить в свойство `server` компонента `MBeanExporter`, чтобы указать, какой сервер `MBean` должен использоваться для экспортования управляемых компонентов `MBean`:

```
<bean id="mbeanExporter"
      class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="spitter:name=HomeController"
              value-ref="homeController"/>
      </map>
    </property>

    <property name="server" ref="mbeanServer" />
</bean>
```

После настройки компонента `MBeanExporter` компонент `homeController` будет экспортиться как Model `MBean` в сервер `MBean` и станет доступен для управления под именем `HomeController`. На рис. 14.2 показано, как выглядит управляемый компонент `homeController` в инструменте `JConsole`.

Слева на рис. 14.2 можно видеть все общедоступные члены компонента `homeController`, экспортимые как атрибуты и операции управляемого компонента `MBean`. Вероятно, это не совсем то, что хотелось бы. В действительности нам хотелось всего лишь иметь возможность изменять свойство `spittlesPerPage`. Нам не нужна возможность вызова метода `showHomePage()` или выполнения других операций с компонентом `HomeController`. То есть нам нужен способ, который позволит указать, какие атрибуты и операции должны экспортироваться.

Фреймворк Spring предлагает несколько возможностей, позволяющих обеспечить более точное управление атрибутами и операциями управляемых компонентов `MBean`, включая:

- объявление имен экспортимых/игнорируемых методов;
- определение интерфейса для компонента, содержащего экспортимые методы;
- использование аннотаций для обозначения управляемых атрибутов и операций.

Попробуем применить каждую из этих возможностей, чтобы понять, какая из них лучше подходит для нашего управляемого компонента `HomeController`. Начнем с объявления имен экспортимых методов.

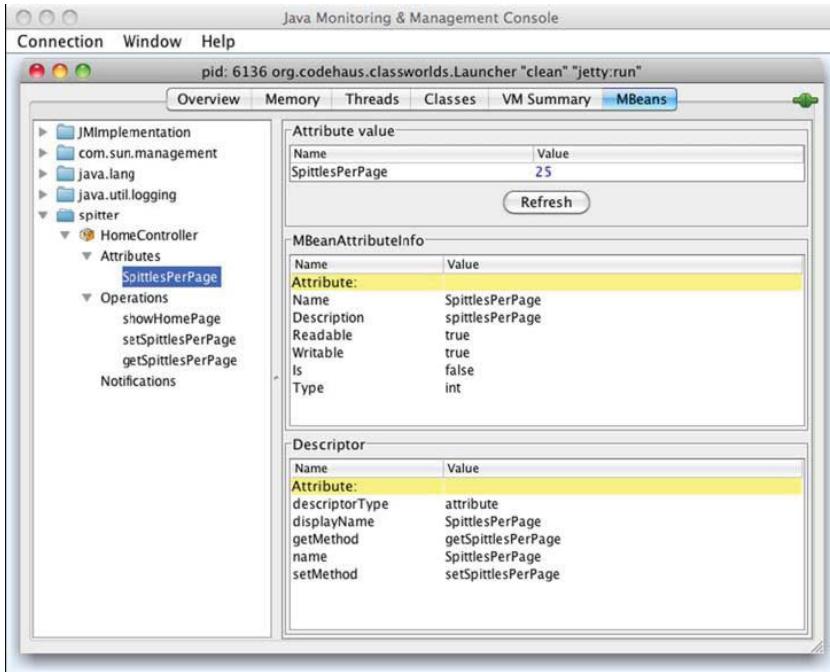


Рис. 14.2. Вид управляемого компонента HomeController в окне JConsole

14.1.1. Экспортирование методов по их именам

Ключом к ограничению набора экспортируемых операций и атрибутов управляемого компонента является *сборщик информации MBean* (MBean info assembler). Одним из таких сборщиков информации является *MethodNameBasedMBeanInfoAssembler*. Ему передается список имен методов, которые должны экспортироваться в виде операций управляемого компонента MBean. В случае с компонентом HomeController нам требуется экспортить атрибут spittlesPerPage. Сможет ли помочь в этом сборщик, экспортирующий методы?

Согласно правилам JavaBean (которые относятся не только к компонентам Spring), атрибут spittlesPerPage может называться свойством только при наличии методов доступа к нему с именами `setSpittlesPerPage()` и `getSpittlesPerPage()`. Чтобы ограничить круг экс-

портируемых атрибутов нашего управляемого компонента MBean, необходимо сообщить компоненту `MethodNameBasedMBeanInfoAssembler`, что интерфейс управляемого компонента будут составлять только эти методы. Следующее объявление компонента `MethodNameBasedMBeanInfoAssembler` содержит только требуемые методы:

```
<bean id="assembler"
      class="org.springframework.jmx.export.assembler.
      ↳MethodNameBasedMBeanInfoAssembler"
      p:managedMethods="getSpittlesPerPage, setSpittlesPerPage" />
```

Свойство `managedMethods` определяет список имен методов, которые должны экспортироваться как операции управляемого компонента MBean. Поскольку эти методы являются методами доступа к свойству `spittlesPerPage`, это приводит к тому, что свойство становится атрибутом управляемого компонента MBean.

Чтобы задействовать компонент `MethodNameBasedMBeanInfoAssembler`, его необходимо внедрить в компонент `MBeanExporter`:

```
<bean id="mbeanExporter"
      class="org.springframework.jmx.export.MBeanExporter">
  <property name="beans">
    <map>
      <entry key="spitter:name=HomeController"
            value-ref="homeController"/>
    </map>
  </property>
  <property name="server" ref="mbeanServer" />

  <property name="assembler" ref="assembler"/>
</bean>
```

Если теперь запустить приложение, для управления будет доступен только атрибут `spittlesPerPage` компонента `HomeController`. На рис. 14.3 показано, как это выглядит в окне JConsole.

Существует еще один сборщик информации, `MethodExclusionMBeanInfoAssembler`, позволяющий ограничивать перечень экспортируемых методов. Этот сборщик является полной противоположностью сборщику `MethodNameBasedMBeanInfoAssembler` – он определяет, какие методы *не должны* экспортироваться как операции управляемого компонента. Например, ниже показано, как с помощью `MethodExclusionMBeanInfoAssembler`

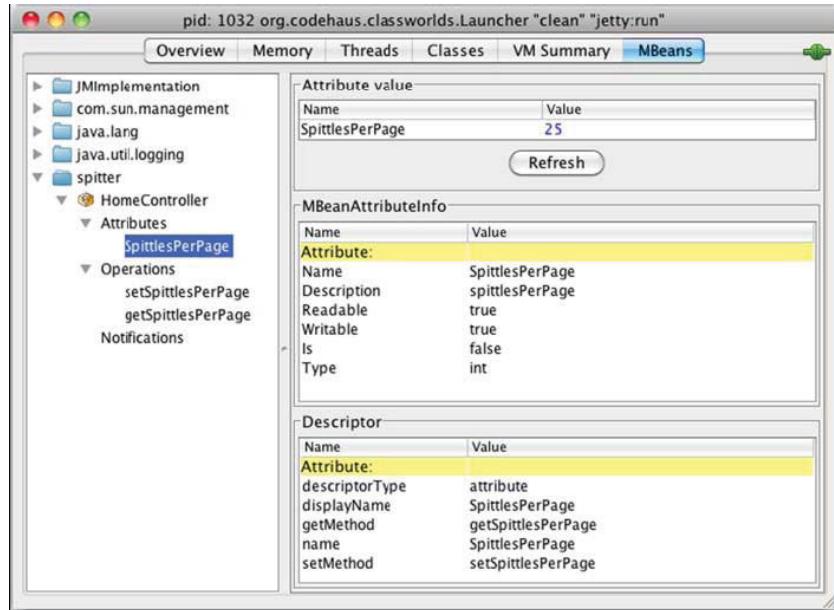


Рис. 14.3. После явного определения экспортируемых методов управляемого компонента HomeController метод `showHomePage()` оказывается недоступным для внешнего мира

оAssembler исключить метод `showHomePage()` из числа экспортируемых операций:

```
<bean id="assembler"
      class="org.springframework.jmx.export.assembler.
      ↳MethodExclusionMBeanInfoAssembler"
      p:ignoredMethods="showHomePage" />
```

Сборщики информации весьма просты в использовании. Но представьте ситуацию, когда необходимо экспортировать несколько компонентов Spring. Список имен методов может стать просто огромным. Кроме того, может так случиться, что потребуется экспортировать некоторый метод одного компонента и исключить экспортацию одноименного метода другого компонента.

Очевидно, что прием, основанный на перечислении имен методов, не годится, когда требуется экспортовать множество управляемых компонентов MBean. Посмотрим, может быть, использование интер-

фейсов, перечисляющих экспортируемые методы, лучше подходит для таких ситуаций.

14.1.2. Определение экспортируемых операций и атрибутов с помощью интерфейсов

Компонент Spring InterfaceBasedMBeanInfoAssembler является еще одним сборщиком информации, который позволяет определять экспортируемые методы с помощью интерфейсов. Он похож на сборщики информации, в которых определяются списки имен экспортируемых методов, за исключением того что вместо списка с именами методов он принимает список интерфейсов.

Например, предположим, что в приложении определен следующий интерфейс HomeControllerManagedOperations:

```
package com.habuma.spitter.jmx;

public interface HomeControllerManagedOperations {
    int getSpittlesPerPage();
    void setSpittlesPerPage(int spittlesPerPage);
}
```

Он определяет методы `setSpittlesPerPage()` и `getSpittlesPerPage` как экспортируемые операции. И снова, так как эти методы являются методами доступа к свойству, их экспортование косвенно приведет к экспортации свойства `spittlesPerPage`. Чтобы задействовать этот сборщик информации, достаточно просто использовать его вместо сборщика со списком методов:

```
<bean id="assembler"
      class="org.springframework.jmx.export.assembler.
      ↗InterfaceBasedMBeanInfoAssembler"
      p:managedInterfaces=
          "com.habuma.spitter.jmx.HomeControllerManagedOperations"
/>
```

Свойство `managedInterfaces` определяет список интерфейсов, которые будут играть роль интерфейсов управляемых компонентов MBean. В данном примере указан единственный интерфейс `HomeControllerManagedOperations`.

Самое интересное, что класс `HomeController` не обязан явно поддерживать интерфейс `HomeControllerManagedOperations`. Этот интерфейс определяется исключительно ради компонента, выполняющего экспортацию, и его не требуется явно реализовывать в программном коде.

Вся прелесть интерфейсов, определяющих экспортируемые операции, состоит в том, что они позволяют собрать десятки методов в нескольких интерфейсах и сохранить определение `InterfaceBasedMBeanInfoAssembler` простым и понятным. Этого вполне достаточно, чтобы упростить конфигурацию Spring, даже при экспортировании множества управляемых компонентов MBean.

Однако так или иначе экспортируемые операции должны быть объявлены где-то в конфигурации Spring или в интерфейсе. Кроме того, объявление экспортируемых операций ведет к дублированию кода – имена методов повторяются в определении интерфейса или в конфигурации контекста Spring, и в программном коде реализации. Это повторение необходимо исключительно для `MBeanExporter`.

УстраниТЬ такое дублирование можно с помощью аннотаций. Посмотрим, как аннотировать компоненты Spring, чтобы обеспечить их экспортацию как управляемых компонентов MBean.

14.1.3. Объявление управляемых компонентов с помощью аннотаций

Помимо сборщиков информации, с которыми мы познакомились выше, в Spring имеется еще один сборщик, известный как `MetadataMBeanInfoAssembler`, который можно настроить для использования аннотаций, отмечающих методы компонентов как операции и атрибуты управляемых компонентов. Я мог бы показать, как пользоваться этим сборщиком, но не буду, так как связывание его вручную – весьма утомительное занятие, не стоящее того, чтобы пользоваться этим способом включения поддержки аннотаций.

Вместо этого я познакомлю вас с элементом `<context:mbean-export>` из конфигурационного пространства имен `context` в Spring. Этот элемент автоматически выполняет связывание экспортёра управляемых компонентов MBean со всеми сборщиками информации, включая поддержку аннотаций Spring. Все, что необходимо сделать, – использовать этот элемент вместо объявления компонента `MBeanExporter`:

```
<context:mbean-export server="mbeanServer" />
```

Теперь, чтобы превратить любой компонент Spring в управляемый компонент MBean, достаточно лишь снабдить его аннотацией `@ManagedResource` и отметить экспортируемые методы аннотацией `@ManagedOperation` или `@ManagedAttribute`. Например, в листинге 14.1 показано, как с помощью представленных аннотаций превратить `HomeController` в управляемый компонент MBean.

Листинг 14.1. Преобразование `HomeController` в управляемый компонент с помощью аннотаций

```
package com.habuma.spitter.mvc;
import java.util.Map;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jmx.export.annotation.ManagedAttribute;
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import com.habuma.spitter.service.SpitterService;

@Controller
@ManagedResource(objectName="spitter:name=HomeController") // Экспортирует
// HomeController
// как MBean
public class HomeController {

    ...
    @ManagedAttribute // Экспортирует spittlesPerPage
                      // как управляемый атрибут
    public void setSpittlesPerPage(int spittlesPerPage) {
        this.spittlesPerPage = spittlesPerPage;
    }

    @ManagedAttribute // Экспортирует spittlesPerPage
                      // как управляемый атрибут
    public int getSpittlesPerPage() {
        return spittlesPerPage;
    }
}
```

Аннотация `@ManagedResource` применяется на уровне класса и указывает, что этот компонент должен экспортиться как MBean. Атрибут `objectName` определяет домен (`spitter`) и имя (`HomeController`) управляемого компонента MBean.

Оба метода доступа к свойству `spittlesPerPage` отмечены аннотацией `@ManagedAttribute`, указывающей, что это свойство должно экспортироваться как атрибут управляемого компонента. Обратите внимание, что необязательно аннотировать оба метода доступа. Если отметить аннотацией только метод `setSpittlesPerPage()`, вы сможете изменять значение свойства посредством механизма JMX, но не сможете просматривать его. Аналогично, если отметить аннотацией только метод `getSpittlesPerPage()`, вы сможете лишь просматривать значение свойства.

Отметьте также, что к методам доступа вместо аннотации `@ManagedAttribute` можно применять аннотацию `@ManagedOperation`. Например:

```
@ManagedOperation  
public void setSpittlesPerPage(int spittlesPerPage) {  
    this.spittlesPerPage = spittlesPerPage;  
}  
  
@ManagedOperation  
public int getSpittlesPerPage() {  
    return spittlesPerPage;  
}
```

В результате эти методы будут экспортироваться как операции управляемого компонента, но при этом свойство `spittlesPerPage` не станет атрибутом управляемого компонента. Это обусловлено тем, что методы, отмеченные аннотацией `@ManagedOperation`, интерпретируются исключительно как обычные методы, а не как методы доступа к свойству. То есть аннотация `@ManagedOperation` должна использоваться только для экспортирования операций управляемого компонента MBean, а аннотация `@ManagedAttribute` – атрибутов.

14.1.4. Разрешение конфликтов между управляемыми компонентами

К настоящему моменту мы познакомились с несколькими способами экспортирования управляемых компонентов MBean. Во всех случаях мы указывали имя MBean, состоящее из имени домена управления и пары ключ/значение. Если предположить, что перед экспортированием MBean его имя еще не использовалось, то у нас не должно возникать никаких проблем. Но что, если такое имя уже использовалось?

По умолчанию MBeanExporter возбудит исключение InstanceAlreadyExistsException, если управляемый компонент с таким именем уже был экспортирован. Однако такое поведение экспортера можно изменить, определив порядок разрешения конфликтов с помощью свойства registrationBehaviorName компонента MBeanExporter или атрибута registration элемента <context:mbean-export>.

Существуют три способа разрешения конфликтов имен, присваиваемых управляемым компонентам MBean:

- ❑ возбудить исключение, если управляемый компонент MBean с таким именем уже существует (поведение по умолчанию);
- ❑ игнорировать конфликт и не экспортить новый управляемый компонент MBean;
- ❑ заменить существующий управляемый компонент MBean новым.

Например, при использовании компонента MBeanExporter его можно настроить так, чтобы он игнорировал конфликты, указав в свойстве registrationBehaviorName значение REGISTRATION_IGNORE_EXISTING, как показано ниже:

```
<bean id="mbeanExporter"
      class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="spitter:name=HomeController"
              value-ref="homeController"/>
      </map>
    </property>
    <property name="server" ref="mbeanServer" />

    <property name="assembler" ref="assembler"/>
    <property name="registrationBehaviorName"
              value="REGISTRATION_IGNORE_EXISTING" />
</bean>
```

В свойстве registrationBehaviorName можно указать значение REGISTRATION_FAIL_ON_EXISTING, REGISTRATION_IGNORE_EXISTING или REGISTRATION_REPLACE_EXISTING, каждое из которых определяет один из трех возможных способов разрешения конфликтов.

При использовании элемента <context:mbean-export>, позволяющего экспортовать управляемые компоненты MBean с помощью аннотаций, метод разрешения конфликтов определяется его атрибутом registration. Например:

```
<context:mbean-export server="mbeanServer"  
registration="replaceExisting"/>
```

В атрибуте `registration` можно указать значение `failOnExisting`, `ignoreExisting` или `replaceExisting`.

Теперь, после регистрации управляемых компонентов MBean с помощью MBeanExporter, необходим некоторый способ, обеспечивающий доступ к ним. Как было показано выше, доступ к локальному серверу MBean и управление компонентами можно осуществлять с помощью таких инструментов, как JConsole. Но эти инструменты не позволяют управлять компонентами MBean программно. Как же тогда реализовать управление компонентами MBean из другого приложения? К счастью, существует другой способ обращения к управляемым компонентам как к удаленным объектам. Посмотрим, как поддержка удаленных управляемых компонентов MBean, имеющаяся в Spring, позволяет обращаться к ним стандартным способом, через механизмы удаленных взаимодействий.

14.2. Удаленные компоненты MBean

Оригинальная спецификация JMX предполагает возможность удаленного управления приложениями посредством управляемых компонентов MBean, но она не определяет фактического протокола удаленных взаимодействий или API. Соответственно, производители реализаций JMX определяют собственные, часто закрытые решения удаленных взаимодействий для JMX.

В ответ на необходимость иметь стандартный механизм удаленных JMX-взаимодействий организация Java Community Process выработала спецификацию *JSR-160*, «Java Management Extensions Remote API Specification» (Спецификация удаленного API расширений управления Java). Данная спецификация определяет стандартный удаленный API для JMX, требующий как минимум использования RMI и дополнительно – протокола обмена сообщениями JMX (JMX Messaging Protocol, JMXMP).

В этом разделе мы познакомимся с поддержкой удаленных управляемых компонентов MBean в Spring. Для начала настроим экспортацию HomeController как удаленного управляемого компонента MBean. А затем посмотрим, как с помощью Spring организовать управление этим удаленным компонентом MBean.

14.2.1. Экспортирование удаленного компонента MBean

Самое простое, что можно сделать, чтобы обеспечить удаленный доступ к управляемому компоненту MBean, – настроить компонент Spring `ConnectorServerFactoryBean`:

```
<bean class="org.springframework.jmx.support.ConnectorServerFactoryBean" />
```

Компонент `ConnectorServerFactoryBean` создает и запускает сервер `JMXConnectorServer`, определяемый спецификацией JSR-160. По умолчанию сервер использует протокол JMXMP и прослушивает порт 9875, то есть он привязан к URL `service:jmx:jmxmp://localhost:9875`. Но мы не ограничены исключительно протоколом JMXMP.

В зависимости от реализации JMX в вашем распоряжении на выбор может иметься несколько протоколов удаленных взаимодействий, включая RMI, SOAP, Hessian/Burlap и даже ПОР. Чтобы определить другой протокол взаимодействий с удаленным компонентом MBean, достаточно просто определить свойство `serviceUrl` компонента `ConnectorServerFactoryBean`. Например, чтобы задействовать протокол RMI для связи с удаленным компонентом MBean, можно определить свойство `serviceUrl`, как показано ниже:

```
<bean class="org.springframework.jmx.support.ConnectorServerFactoryBean"  
      p:serviceUrl="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/spitter" />
```

Здесь выполняется связывание с реестром RMI, действующим на локальном компьютере и обслуживающим порт 1099. Это также означает необходимость запустить реестр RMI, указав ему этот номер порта. Как рассказывалось в главе 11, реестр RMI можно запускать автоматически, с помощью компонента `RmiServiceExporter`. Но в данном случае компонент `RmiServiceExporter` не используется, поэтому реестр RMI следует запустить вручную, объявив компонент `RmiRegistryFactoryBean`, как показано ниже:

```
<bean class="org.springframework.remoting.rmi.RmiRegistryFactoryBean"  
      p:port="1099" />
```

Вот и все! Теперь все наши управляемые компоненты MBean будут доступны посредством RMI. Но осталось еще кое-что – пока еще

не реализована возможность обращения к удаленным управляемым компонентам MBean посредством RMI. Поэтому обратим свое внимание на сторону клиента и посмотрим, как связаться с удаленным компонентом MBean с помощью Spring.

14.2.2. Доступ к удаленным компонентам MBean

Организация доступа к удаленному серверу MBean связана с настройкой MBeanServerConnectionFactoryBean в контексте Spring. Следующее объявление определяет настройки компонента MBeanServerConnectionFactoryBean, который можно использовать для доступа к удаленному RMI-серверу, созданному в предыдущем разделе:

```
<bean id="mBeanServerClient"
      class=
        "org.springframework.jmx.support.MBeanServerConnectionFactoryBean"
      p:serviceUrl=
        "service:jmx:rmi://localhost/jndi/rmi://localhost:1099/spitter"/>
```

Как следует из имени компонента MBeanServerConnectionFactoryBean, это фабричный компонент, создающий объекты MBeanServerConnection. Объекты MBeanServerConnection действуют как локальные прокси-объекты, обеспечивающие возможность взаимодействий с удаленным сервером MBean. Его можно внедрять в свойства компонентов, как любой другой компонент:

```
<bean id="jmxClient" class="com.springinaction.jmx.JmxClient">
  <property name="mbeanServerConnection" ref="mBeanServerClient" />
</bean>
```

Объект MBeanServerConnection предоставляет несколько методов, позволяющих обращаться к удаленному серверу MBean и вызывать методы управляемых компонентов, содержащихся в нем. Например, допустим, что нам необходимо определить количество управляемых компонентов, зарегистрированных в удаленном сервере MBean. Следующий фрагмент кода выведет эту информацию:

```
int mbeanCount = mbeanServerConnection.getMBeanCount();
System.out.println("There are " + mbeanCount + " MBeans");
```

У удаленного сервера можно также запросить имена всех управляемых компонентов MBean, воспользовавшись методом queryNames():

```
java.util.Set mbeanNames = mbeanServerConnection.queryNames(null, null);
```

Два параметра, передаваемые методу `queryNames()`, используются для фильтрации результатов. Если передать значение `null` в обоих параметрах, метод вернет имена всех зарегистрированных управляемых компонентов.

Возможность запросить у удаленного сервера MBean количество и имена управляемых компонентов сама по себе интересна, но не имеет большой практической ценности. Настоящую ценность имеет возможность обращения к атрибутам и методам управляемых компонентов, зарегистрированных в удаленном сервере MBean.

Для доступа к атрибутам управляемых компонентов можно использовать методы `getAttribute()` и `setAttribute()`. Например, чтобы получить значение атрибута управляемого компонента, можно вызвать метод `getAttribute()`, как показано ниже:

```
String cronExpression = mbeanServerConnection.getAttribute(  
    new ObjectName("spitter:name=HomeController"), "spittlesPerPage");
```

Аналогично изменить значение атрибута управляемого компонента можно с помощью метода `setAttribute()`:

```
mbeanServerConnection.setAttribute(  
    new ObjectName("spitter:name=HomeController"),  
    new Attribute("spittlesPerPage", 10));
```

Если потребуется вызывать операцию управляемого компонента, метод `invoke()` – это то, что вам нужно. Ниже показано, как вызвать метод `setSpittlesPerPage()` управляемого компонента `HomeController`:

```
mbeanServerConnection.invoke(  
    new ObjectName("spitter:name=HomeController"),  
    "setSpittlesPerPage",  
    new Object[] { 100 },  
    new String[] {"int"});
```

С помощью методов объекта `MBeanServerConnection` можно выполнять десятки других операций с удаленными компонентами MBean. Оставляю исследование этих возможностей в качестве самостоятельного упражнения.

Но вызов методов и изменение значений атрибутов удаленных компонентов MBean выполняется очень неудобно, когда эти опе-

рации производятся посредством MBeanServerConnection. Даже вызов такого простого метода, как setSpittlesPerPage(), сопряжен с необходимостью создания экземпляра ObjectName и передачи дополнительных параметров методу invoke(). Обращение к методам управляемых компонентов выглядит намного сложнее, чем обращение к обычным методам. Более простой способ обращения к удаленным компонентам MBean предоставляют прокси-объекты.

14.2.3. Проксирование управляемых компонентов

Компонент Spring MBeanProxyFactoryBean – это фабричный компонент, создающий прокси-объекты и действующий подобно фабричным компонентам, рассматривавшимся в главе 11. Только создаваемые им прокси-объекты обеспечивают доступ не к удаленным службам Spring, а к удаленным управляемым компонентам MBean (как если бы они были локальными компонентами). На рис. 14.4 показано, как действует этот механизм.

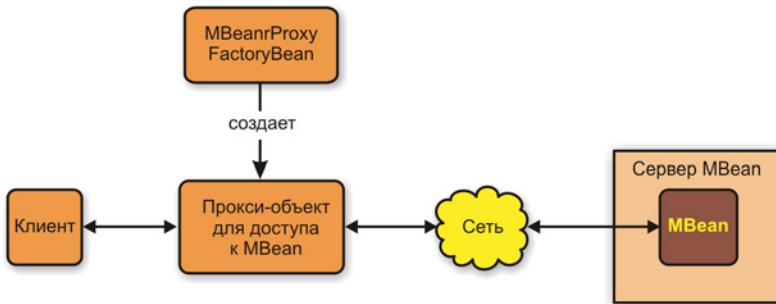


Рис. 14.4. MBeanProxyFactoryBean создает прокси-объекты для логистика к удаленным компонентам MBean. Благодаря прокси-объекту клиент может взаимодействовать с удаленным компонентом MBean, как если бы он был локально настроенным POJO

Например, взгляните на следующее объявление компонента MBeanProxyFactoryBean:

```
<bean id="remoteHomeControllerMBean"
      class="org.springframework.jmx.access.MBeanProxyFactoryBean"
      p:objectName="spitter:name=HomeController"
```

```
p:server-ref="mBeanServerClient"
p:proxyInterface=
    "com.habuma.spitter.jmx.HomeControllerManagedOperations" />
```

Свойство `objectName` определяет имя удаленного компонента MBean. Здесь это свойство ссылается на компонент MBean `HomeController`, экспортированный ранее.

Свойство `server` ссылается на компонент `MBeanServerConnection`, через который будут осуществляться все взаимодействия. Здесь в это свойство внедряется ссылка на компонент `MBeanServerConnectionFactory`, настроенный выше.

Наконец, свойство `proxyInterface` определяет интерфейс, который будет реализован прокси-объектом. В данном случае используется интерфейс `HomeControllerManagedOperations`, который был определен в разделе 14.1.2.

После объявления компонента `remoteHomeControllerMBean` его можно внедрить в любой другой компонент, в свойство типа `HomeControllerManagedOperations`, и использовать для доступа к удаленному компоненту MBean, то есть вызывать его методы `setSpittlesPerPage()` и `getSpittlesPerPage()`.

Итак, мы познакомились с несколькими способами организации взаимодействий с управляемыми компонентами MBean и теперь можем просматривать и изменять настройки компонентов Spring прямо во время выполнения приложения. Однако до сих пор эти взаимодействия носили односторонний характер. Мы могли обращаться к компонентам MBean, а компоненты MBean не могли обращаться к внешнему миру. Теперь пришло время послушать, что они могут сообщить нам, обеспечив прием извещений от них.

14.3. Обработка извещений

Обращение к управляемому компоненту MBean за информацией – это лишь один из способов слежения за состоянием приложения. Но он оказывается не самым эффективным, когда требуется получать информацию о важных событиях, происходящих внутри приложения.

Например, представьте, что в приложении Spitter необходимо постоянно вести учет количества отправленных сообщений. И требуется сразу же определять каждое миллионное сообщение. Для этого можно написать некоторый программный код, который периодиче-

ски будет выполнять запрос к базе данных, возвращающий количество сообщений. Но процесс, выполняющий такой запрос, будет создавать лишнюю нагрузку на базу данных из-за необходимости постоянной проверки количества сообщений.

Вместо постоянных обращений к базе данных предпочтительнее было бы иметь возможность для управляемых компонентов MBean извещать приложение о важных событиях в моменты, когда они происходят. Извещения JMX, изображенные на рис. 14.5, как раз являются такой возможностью, позволяющей управляемым компонентам активно взаимодействовать с внешним миром, вместо того чтобы ждать, пока какое-нибудь приложение не запросит нужную ему информацию.

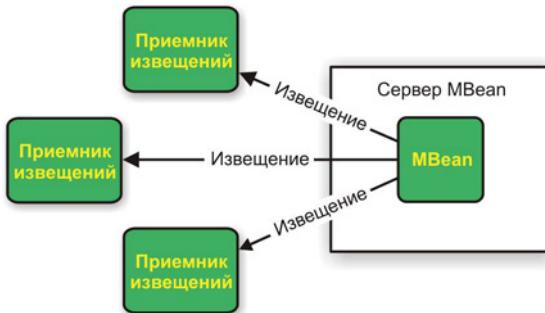


Рис. 14.5. Извещения JMX
позволяют управляемым компонентам MBean
активно взаимодействовать с внешним миром

Поддержка отправки извещений реализована в фреймворке Spring в форме интерфейса `NotificationPublisherAware`. Любой компонент, превращаемый в управляемый компонент MBean, может реализовать этот интерфейс и получить возможность отправлять извещения. Например, в листинге 14.2 представлен класс `SpittleNotifierImpl`.

Листинг 14.2. Использование интерфейса `NotificationPublisherAware` для отправки извещений JMX

```

package com.habuma.spitter.jmx;

import javax.management.Notification;
import org.springframework.jmx.export.annotation.ManagedNotification;
  
```

```
import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.notification.NotificationPublisher;
import org.springframework.jmx.export.notification.*;
import NotificationPublisherAware;
import org.springframework.stereotype.Component;

@Component
@ManagedResource("spitter:name=SpitterNotifier")
@ManagedNotification(
    notificationTypes="SpittleNotifier.OneMillionSpittles",
    name="TODO")

// Реализует интерфейс NotificationPublisherAware
public class SpittleNotifierImpl
    implements NotificationPublisherAware, SpittleNotifier {

    private NotificationPublisher notificationPublisher;

    public void setNotificationPublisher( // Внедряется NotificationPublisher
        NotificationPublisher notificationPublisher) {
        this.notificationPublisher = notificationPublisher;
    }

    public void millionthSpittlePosted() {
        notificationPublisher.sendNotification( // Отправка сообщения
            new Notification(
                "SpittleNotifier.OneMillionSpittles", this, 0));
    }
}
```

Как видите, класс SpittleNotifierImpl реализует интерфейс NotificationPublisherAware. Это не слишком сложный интерфейс. Он требует реализовать всего один метод: setNotificationPublisher.

Класс SpittleNotifierImpl реализует также единственный метод интерфейса SpittleNotifier¹, millionthSpittlePosted(). Для отправки сообщения об очередном миллионном сообщении этот метод использует компонент NotificationPublisher, который автоматически внедряется посредством метода setNotificationPublisher().

Как только будет вызван метод sendNotification(), сообщение отправится в... хм-м-м... похоже, что мы еще не определили, куда от-

¹ Для краткости я опустил интерфейс SpittleNotifier. Но, как можно догадаться, его единственным методом является метод millionthSpittlePosted.

правится извещение. Настроим приемник извещений, который будет получать извещения и реагировать на них.

14.3.1. Прием извещений

Стандартный способ организовать прием извещений от управляемых компонентов MBean – реализовать интерфейс `javax.management.NotificationListener`. Например, взгляните на следующую реализацию класса `PagingNotificationListener`:

```
package com.habuma.spitter.jmx;
import javax.management.Notification;
import javax.management.NotificationListener;

public class PagingNotificationListener implements NotificationListener {
    public void handleNotification(Notification notification,
                                    Object handback) {
        // ...
    }
}
```

Класс `PagingNotificationListener` – это типичный приемник извещений JMX. При получении извещения вызывается его метод `handleNotification()`. Вероятно, метод `handleNotification()` класса `PagingNotificationListener` будет посыпать сообщение на пейджер или сотовый телефон, извещаю об очередном миллионном сообщении. (Фактическую реализацию метода я оставляю на усмотрение читателя.)

Единственное, что осталось сделать, – зарегистрировать компонент `PagingNotificationListener` с помощью компонента `MBeanExporter`:

```
<bean class="org.springframework.jmx.export.MBeanExporter">
    ...
    <property name="notificationListenerMappings">
        <map>
            <entry key="Spitter:name=PagingNotificationListener">
                <bean
                    class="com.habuma.spitter.jmx.PagingNotificationListener" />
            </entry>
        </map>
    </property>
</bean>
```



Свойство `notificationListenerMappings` компонента `MBeanExporter` используется для внедрения приемников извещений, поступающих от управляемых компонентов. В данном случае компонент `PagingNotificationListener` настраивается на прием любых извещений, отправляемых управляемым компонентом `SpittleNotifier`.

14.4. В заключение

С помощью механизма JMX можно заглянуть внутрь действующего приложения. В этой главе было показано, как настроить фреймворк Spring, чтобы он автоматически экспортировал компоненты как управляемые компоненты MBean с поддержкой JMX, с целью получения доступа к ним с помощью инструментов управления, поддерживающих механизм JMX. Здесь также было показано, как создавать и использовать удаленные управляемые компоненты MBean, когда эти компоненты и инструменты управления отделены друг от друга. Наконец, было показано, как с помощью фреймворка Spring организовать отправку и прием извещений JMX.



Глава 15. Создание веб-служб на основе модели contract-first

В этой главе рассматриваются следующие темы:

- создание XML-определений служб;
- создание веб-служб, ориентированных на документы;
- маршалинг и демаршалинг XML-сообщений;
- создание клиентов веб-служб на основе шаблонов.

Представьте, что на эти выходные у вас запланирована поездка. Прежде чем пуститься в путь, нужно зайти в банк, положить чек с зарплатой и получить немного наличных на расходы.

Это самая обычная череда событий, единственное, что в ней необычного, – это банк. Когда вы заходите внутрь, вы не видите банковских служащих, готовых помочь вам. Вместо этого вы получаете возможность выполнить все перечисления денег самостоятельно. У вас есть прямой доступ к программе учета и к хранилищу, и вам позволено самому выполнить все необходимые операции. Итак, вы выполняете следующие действия:

1. Помещаете подписанный чек с заработной платой в коробку для обналиченных чеков.
2. В бухгалтерской книге исправляете свой баланс, увеличивая его на сумму в чеке.
3. Забираете из хранилища \$200 и кладете в свой кошелек.
4. В бухгалтерской книге исправляете свой баланс, уменьшая его на \$200.
5. В качестве благодарности за проделанную работу вы платите самому себе комиссионные, положив \$50 на другой счет, и выходите за дверь.

Стоп! С шагами 1–4 вроде бы все в порядке. Но шаг 5 выглядит немного странным.

Проблема (если это можно назвать проблемой) с этим банком в том, что он доверяет своим клиентам слишком полный доступ



к внутренним операциям. Вместо того чтобы предоставить интерфейс к внутренним операциям (обычно известный под названием «кассир»), банк дает полный доступ к внутренним операциям, позволяя вам поступать по собственному усмотрению. Как следствие у клиента появляется возможность взять деньги из хранилища, не зафиксировав это.

Как бы здорово это ни было для клиентов, большинство банков действуют совершенно иначе (если вы знаете банк, который дает такой же полный доступ к внутренним операциям, сообщите мне – я очень хотел бы открыть там счет!). В большинстве банков есть кассиры, банкоматы и веб-сайты, позволяющие управлять счетом. Все эти интерфейсы доступа к банковским операциям являются абстракциями хранилища и бухгалтерской книги. Они с улыбкой оказывают вам услуги, но позволяют выполнять только те операции, которые укладываются в бизнес-модель банка.

Аналогично большинство приложений не позволяют обращаться к внутренним объектам напрямую. Возьмем в качестве примера веб-приложения. Как было показано в главе 8, пользователи взаимодействуют с веб-приложением на основе Spring MVC посредством контроллеров. В недрах приложения могут быть скрыты десятки или даже сотни объектов, выполняющих основные операции в приложении. Но пользователю позволено взаимодействовать только с контроллерами, которые, в свою очередь, взаимодействуют с внутренними объектами.

В главе 11 было показано, как легко и просто можно создавать веб-службы с использованием различных экспортёров. Но, экспортируя компонент приложения как веб-службу, мы экспортируем внутренний API приложения, что влечет за собой определенные последствия: по аналогии с примером организации банковских услуг, представленным выше, необходимо проявить особую осторожность, чтобы не открыть слишком широкий доступ к внутреннему API. В противном случае клиенты веб-службы могут получить доступ к внутренним операциям, которые им не нужны.

В этой главе будет представлен альтернативный способ конструирования веб-служб с помощью фреймворка Spring-WS. Мы отделим определение внешнего API службы от внутреннего и сосредоточимся на обмене сообщениями между клиентами и службой, а не на вызове удаленных методов.

Не буду обманывать: создание веб-служб на основе фреймворка Spring-WS выглядит гораздо сложнее, чем экспортование компо-

нентов с помощью JAX-WS. Однако, как мне кажется, это не намного сложнее, а архитектурные преимущества, которые предлагает Spring-WS, заслуживают внимания.

15.1. Введение в Spring-WS

Фреймворк Spring Web Services (или Spring-WS) – один из проектов, развивающихся в рамках Spring, целью которого является создание веб-служб на основе модели «contract-first». Что же это за модель «contract-first»? Ответить на этот вопрос будет проще, если сначала рассмотреть противоположную ей модель организации веб-служб: «contact-last».

В главе 11 (см. раздел 11.5.1) мы использовали JAX-WS для экспортования компонентов как удаленных веб-служб. Сначала мы написали программный код на языке Java (реализацию службы). Затем определили соответствующий компонент в конфигурации Spring. И наконец, воспользовались классом `SimpleJaxWsServiceExporter`, чтобы превратить его в веб-службу. Мы не объявляли явно определение службы (WSDL и XSD). Механизм JAX-WS автоматически генерирует это определение *после* развертывания службы. Проще говоря, определение веб-службы создавалось в последнюю очередь, что в точности соответствует значению названия «contract-last».

Модель «contract-last» разработки веб-служб пользуется большой популярностью по одной основной причине: простота. Большинство разработчиков не обладают стойкостью духа, необходимой для освоения WSDL, SOAP и XML Schema (XSD). В модели «contract-last» нет необходимости разбираться со сложными файлами WSDL и XSD. Достаточно лишь написать Java-класс, реализующий службу, и попросить фреймворк «SOAP-ифизировать» его. Если фреймворк, такой JAX-WS, способен решить эту проблему, зачем тогда нам беспокоиться о ней?

Но есть одна маленькая проблема: при использовании модели разработки веб-службы «contract-last» ее определение является прямым отражением внутреннего API приложения. Как правило, внутренний API приложения является намного более изменчивым, чем допустимо для внешнего API. А при использовании модели «contract-last» изменение внутреннего API влечет за собой изменение определения службы, что в конечном итоге может потребовать внести изменения в клиентские приложения, пользующиеся служ-

бой. Даже простой рефакторинг, выполненный сегодня, может привести к изменению определения службы завтра.

Все это приводит к классической проблеме поддержки нескольких версий веб-служб. Беда в том, что изменить определение службы намного проще, чем изменить реализацию клиентов, использующих эту службу. Если веб-службой пользуются 1000 клиентов, то после изменения определения службы 1000 клиентов окажутся неработоспособны, пока не будут изменены в соответствии с новым определением. Типичное решение этой проблемы состоит в том, чтобы поддерживать несколько версий службы, пока все клиенты не будут обновлены. Однако это многократно увеличивает стоимость обслуживания и поддержки, так как вы вынуждены будете поддерживать несколько версий одной и той же службы.

Гораздо проще стараться избегать изменения определения службы. А когда такие изменения необходимы, они не должны нарушать совместимость с предыдущими версиями. Но этого сложно добиться, когда определение службы генерируется автоматически.

Проще говоря, проблема в том, что в модели «*contract-last*» самый важный элемент, определение службы, интерпретируется как нечто второстепенное. Основное внимание в этой модели уделяется *особенностям реализации* веб-службы, а не тому, что она должна делать.

Решение данной проблемы состоит в том, чтобы перевернуть все с головы на ноги – создать сначала определение веб-службы, а затем решать вопросы реализации. Такая модель разработки называется «*contract-first*». Определение веб-службы составляется почти без учета, как должно выглядеть приложение, реализующее его. Это достаточно практично, потому что такой подход подчеркивает – *что ожидается от службы, а не как она должна быть реализована*.

Что? Почувствовали себя некомфортно. Может быть, вы съели на обед необычно большой бурритто... или, может быть, вас испугала необходимость писать WSDL-файл вручную?

Не волнуйтесь. Я не настолько бесчеловечен, как могло показаться. В процессе повествования я познакомлю вас с некоторыми приемами, упрощающими определение службы. (Если вам не стало легче, советую принять антацидное средство и в будущем сократить в своем рационе количество острых блюд.)

Основной рецепт разработки веб-служб с использованием модели «*contract-first*» и на основе фреймворка Spring-WS представлен в табл. 15.1.

Таблица 15.1. Этапы разработки веб-службы в модели «contract-first»

Этап	Действие	Описание
1	Составить определение службы	На этой стадии проектируются примеры сообщений XML, которые будут обрабатываться веб-службой. Эти примеры сообщений будут использоваться в создании схемы XML Schema для последующего создания WSDL-файла
2	Написать конечную точку службы	На этой стадии создаются классы, принимающие и обрабатывающие сообщения, отправляемые веб-службе
3	Настроить конечную точку и инфраструктуру Spring-WS	На этой стадии производится связывание конечной точки веб-службы и компонентов Spring-WS

Для демонстрации особенностей разработки веб-служб на основе фреймворка Spring создадим службу, оценивающую комбинацию карт при игре в покер. На рис. 15.1 изображены требования, предъявляемые к веб-службе: по пяти картам необходимо идентифицировать их значение при игре в покер.

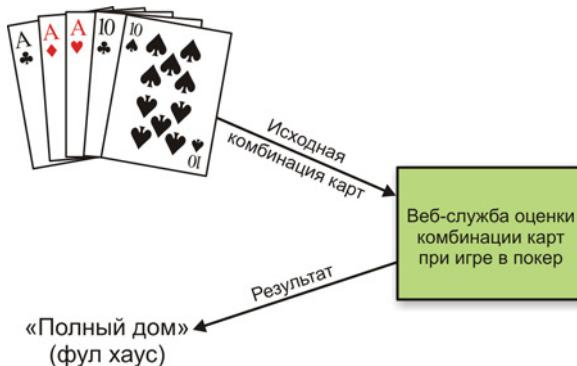


Рис. 15.1. Служба оценки комбинации карт при игре в покер.

Службе передается комбинация из пяти карт, а она определяет их значение при игре в покер

Поскольку разработка ведется с использованием модели «contract-first», первое, что следует сделать, – создать определение службы. Начнем.



15.2. Определение API службы (в первую очередь!)

Единственный важный этап в разработке веб-службы в модели «contract-first» – определение веб-службы. Определяя службу, мы фактически определяем формат сообщений, посредством которых будет происходить обмен информацией между клиентами и службой, не зависящий от особенностей реализации обработки этих сообщений.

Даже при том, что основной темой этой главы является обсуждение фреймворка Spring-WS, вы увидите, что этот раздел практически никак не связан с фреймворком Spring. Это обусловлено тем, что внешний API веб-службы должен определяться независимо от фактической его реализации. Основное внимание на этом этапе должно уделяться не тому, *как* будет реализована служба, а *что* она должна делать. К объединению с фреймворком Spring-WS мы приступим в разделе 15.3. А в этом разделе будут описываться приемы, которые применимы к модели разработки веб-служб «contract-first» вообще, независимо от используемого фреймворка.

Основное внимание в модели «contract-first» разработки веб-служб уделяется сообщениям, отправляемым веб-службам и принимаемым от них. Поэтому первым шагом на пути определения службы является определение формата сообщений. Для начала определим примеры XML-сообщений, которые затем будем использовать для определения службы.

15.2.1. Создание примеров XML-сообщений

Говоря простым языком, наша служба будет принимать комбинацию из пяти карт и возвращать ее оценку при игре в покер (например, фул-хаус, флеш и т. д.). Сообщение, передаваемое веб-службе, можно представить так:

```
<EvaluateHandRequest
    xmlns="http://www.springinaction.com/poker/schemas">
    <card>
        <suit>HEARTS</suit>
        <face>TEN</face>
    </card>
    <card>
```

```
<suit>SPADES</suit>
<face>KING</face>
</card>
<card>
    <suit>HEARTS</suit>
    <face>KING</face>
</card>
<card>
    <suit>DIAMONDS</suit>
    <face>TEN</face>
</card>
<card>
    <suit>CLUBS</suit>
    <face>TEN</face>
</card>
</EvaluateHandRequest>
```

Выглядит достаточно просто, не находите? Сообщение содержит пять элементов `<card>`, в каждом из которых присутствуют элементы `<suit>` и `<face>`. Сообщение достаточно полно описывает комбинацию карт. Все элементы `<card>` находятся внутри элемента `<EvaluateHandRequest>`, который определяет тип сообщения, отправляемого службе.

Сообщение, возвращаемое в ответ на это сообщение, выглядит еще проще:

```
<EvaluateHandResponse
    xmlns="http://www.springinaction.com/poker/schemas">
    <handName>Full House</handName>
</EvaluateHandResponse>
```

Элемент `<EvaluateHandResponse>`, определяющий тип сообщения, содержит единственный элемент `<handName>`, хранящий значение комбинации карт.

Эти примеры сообщений будут служить основой определения нашей службы. Кто-то может не поверить в это, но, определив примеры сообщений, мы преодолели самый сложный этап на пути определения службы. Без шуток.

Определение формата представления данных

Теперь можно приступать к созданию определения службы. Однако прежде разделим определение службы на две части.

- ❑ *Определение формата представления данных* – описывает формат сообщений, получаемых и отправляемых службой. В данном примере под этим подразумевается определение схемы сообщений `<EvaluateHandRequest>` и `<EvaluateHandResponse>`.
- ❑ *Определение перечня операций* – описывает операции, выполняемые службой. Обратите внимание, что методы, составляющие API службы, необязательно должны соответствовать операциям, определяемым протоколом SOAP.

Обе эти части определения обычно (но не всегда) располагаются в едином WSDL-файле. Как правило, в WSDL-файле содержится встроенная схема на языке XML Schema, определяющая формат представления данных. Остальная часть WSDL-файла определяет перечень поддерживаемых операций в виде одного или более элементов `<wsdl:operation>` внутри элемента `<wsdl:binding>`.

Пусть вас не пугает все, что было сказано в предыдущем абзаце. Как я обещал, создание определения службы будет достаточно простым делом, поэтому от вас не потребуется знать все тонкости создания WSDL-файлов. Главное – помнить, что определение службы делится на две основные части.

Формат представления данных определяется с помощью языка XML Schema (XSD). Схема XSD позволяет точно определить, что должно содержаться в сообщении. Мы можем определить не только элементы, встречающиеся в сообщениях, но и типы сообщений, а также ограничения на данные, которые могут передаваться в этих сообщениях.

XSD-файл легко можно создать вручную, но это слишком большой объем работы, по сравнению с тем, что я хотел бы сделать. Поэтому я воспользуюсь специализированным инструментом, который исследует один или более XML-файлов и на основе их содержимого производит файл схемы на языке XML Schema, с помощью которого будет проверяться допустимость XML-файлов сообщений.

Существует множество различных инструментов, позволяющих создавать XSD-файлы, но я предпочитаю использовать Trang. Trang – это инструмент командной строки (его можно получить по адресу: www.thaiopensource.com/relaxng/trang.html), принимающий XML-файл на входе и производящий XSD-файл на выходе (рис. 15.2). Инструмент Trang написан на языке Java и потому может использоваться везде, где установлена виртуальная машина Java (JVM). Как можно предположить из URL, инструмент Trang позволяет генерировать схемы на языке RELAX NG (альтернативный

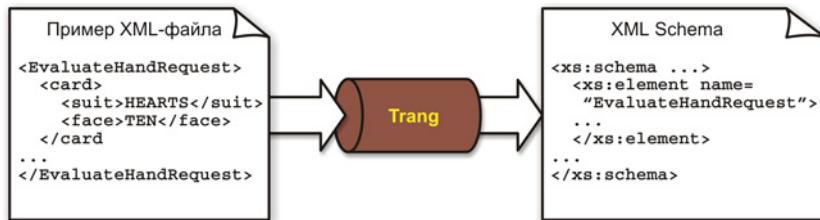


Рис. 15.2. Trang – это инструмент, производящий XSD-файлы на основе примеров XML-файлов

язык описания структуры XML-документа), но его также можно использовать для создания файлов на языке XML Schema. Для нужд фреймворка Spring-WS инструмент Trang будет использоваться для создания файлов на языке XML Schema.

Загрузив и распаковав дистрибутив Trang, в каталоге можно обнаружить файл `trang.jar`. Это выполняемый JAR-файл, поэтому, чтобы запустить инструмент Trang, достаточно выполнить команду:

```
% java -jar trang.jar EvaluateHandRequest.xml
↳ EvaluateHandResponse.xml PokerTypes.xsd
```

При запуске инструмента Trang я передал ему три аргумента командной строки. Первые два – файлы с примерами XML-сообщений, созданные выше. Поскольку инструменту Trang было передано оба файла, полученный в результате XSD-файл можно будет использовать для проверки сообщений обоих типов. Последний аргумент – имя XSD-файла, который должен получиться в результате.

При запуске с этими аргументами Trang генерирует определение формата представления данных для нашей службы и сохранит его в файле `PokerTypes.xsd` (листинг 15.1).

Листинг 15.1. Содержимое файла `PokerTypes.xsd`, определяющее формат представления данных для веб-службы

```
<?xml version="1.0" encoding="UTF-8"?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema"
              elementFormDefault="qualified"
              targetNamespace=
                "http://www.springinaction.com/poker/schemas"
              xmlns:schemas=
                "http://www.springinaction.com/poker/schemas">
```

```
<xs:element name="EvaluateHandRequest">
    <xs:complexType>
        <xs:sequence>
            <xs:element maxOccurs="unbounded"
                ref="schemas:card"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="card">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="schemas:suit"/>
            <xs:element ref="schemas:face"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="suit" type="xs:NCName"/>
<xs:element name="face" type="xs:NCName"/>
<xs:element name="EvaluateHandResponse">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="schemas:handName"/>
        </xs:sequence>
    </xs:complexType>
</xs:element>
<xs:element name="handName" type="xs:string"/>
</xs:schema>
```

Инструмент Trang избавляет нас от массы проблем, генерируя XSD-описание для наших сообщений. Но не избавляет от необходимости вручную проверять получившуюся схему. Генерируя описание XSD, инструмент Trang делает некоторые предположения о том, какие данные будут передаваться в XML-сообщениях. В большинстве своем эти предположения дают вполне приемлемые результаты. Но нередко бывает необходимо вручную подправить сгенерированный XSD-файл.

Например, инструмент Trang предполагает, что значения элементов `<suit>` и `<face>` должны определяться как имена, не содержащие двоеточий¹ (`xs:NCName`). В действительности же нам требуется, чтобы

¹ Имена без двоеточий – это имена, которые не содержат уточняющего префикса пространства имен. То есть они не содержат символа двоеточия (:).

значениями этих элементов были простые строки (`xs:string`). Поэтому исправим определения элементов `<suit>` и `<face>`, как показано ниже:

```
<xs:element name="suit" type="xs:string"/>
<xs:element name="face" type="xs:string"/>
```

Нам также известно, что элемент `<suit>` может иметь только четыре значения, поэтому можно наложить дополнительные ограничения:

```
<xs:element name="suit" type="schemas:Suit" />
<xs:simpleType name="Suit">
    <xsd:restriction base="xs:string">
        <xsd:enumeration value="SPADES" />
        <xsd:enumeration value="CLUBS" />
        <xsd:enumeration value="HEARTS" />
        <xsd:enumeration value="DIAMONDS" />
    </xsd:restriction>
</xs:simpleType>
```

Аналогично элемент `<face>` может иметь только 13 значений, поэтому определим эти ограничения в XSD-файле:

```
<xs:element name="face" type="schemas:Face" />
<xs:simpleType name="Face">
    <xsd:restriction base="xs:string">
        <xsd:enumeration value="ACE" />
        <xsd:enumeration value="TWO" />
        <xsd:enumeration value="THREE" />
        <xsd:enumeration value="FOUR" />
        <xsd:enumeration value="FIVE" />
        <xsd:enumeration value="SIX" />
        <xsd:enumeration value="SEVEN" />
        <xsd:enumeration value="EIGHT" />
        <xsd:enumeration value="NINE" />
        <xsd:enumeration value="TEN" />
        <xsd:enumeration value="JACK" />
        <xsd:enumeration value="QUEEN" />
        <xsd:enumeration value="KING" />
    </xsd:restriction>
</xs:simpleType>
```

Обратите также внимание: инструмент Trang ошибочно предполагает, что элемент `<EvaluateHandRequest>` может содержать неограниченное количество элементов `<card>` (`maxOccurs="unbounded"`). Однако при игре в покер в одних руках может содержаться только точно пять карт. Это ограничение также необходимо включить в определение элемента `<EvaluateHandRequest>`:

```
<xss:element name="EvaluateHandRequest">
  <xss:complexType>
    <xss:sequence>
      <xss:element minOccurs="5" maxOccurs="5"
        ref="schemas:card"/>
    </xss:sequence>
  </xss:complexType>
</xss:element>
```

Теперь определение элемента `<EvaluateHandResponse>` достаточно полное. Можно было бы также ограничить допустимые значения, возвращаемые в элементе `<handName>`, но в этом нет необходимости. Поэтому оставим определение данного элемента без изменений.

Мы закончили определение формата представления данных для службы оценки комбинации карт при игре в покер, но как быть с определением перечня операций? Разве нам не потребуется создать некоторый WSDL-файл, полностью определяющий веб-службу?

Да, WSDL-определение совершенно необходимо – в конце концов, WSDL – это стандартный способ определения веб-служб. WSDL-файл можно было бы создать вручную, но в этом занятии мало приятного. И снова я обещал, что мы не будем сталкиваться со сложностями. Но должен просить вас подождать немного, прежде чем я покажу, где нам потребуется определение перечня операций. Создание WSDL-файла будет продемонстрировано в разделе 15.4.6, когда будет рассказываться о связывании компонента, генерирующего WSDL-определение.

Но прежде нам необходимо создать конечную точку службы. Определение службы описывает лишь сообщения, посылаемые службе и получаемые от нее, но не порядок их обработки. Посмотрим далее, как создать конечные точки обработки сообщений в Spring-WS, получаемых от клиента веб-службы.

15.3. Обработка сообщений в веб-службе

Как отмечалось в начале этой главы, хорошо спроектированное приложение не дает прямого доступа к внутренним объектам, реализующим внутренние операции. В Spring MVC, например, пользователь взаимодействует с приложением посредством контроллеров, которые транслируют запросы пользователя в вызовы методов внутренних объектов.

Возможно, вам будет полезно знать, что в этом отношении фреймворки Spring MVC и Spring-WS очень похожи. Пользователи взаимодействуют с приложениями на основе фреймворка Spring MVC посредством контроллеров, а клиенты веб-служб взаимодействуют со службами на основе фреймворка Spring-WS – посредством конечных точек.

На рис. 15.3 показано, как клиенты взаимодействуют с конечными точками. Конечная точка – это класс, принимающий XML-сообщение от клиента и, опираясь на содержимое сообщения, выполняющий вызовы методов внутренних объектов. Конечная точка службы оценки комбинации карт будет обрабатывать сообщения `<EvaluateHandRequest>`.

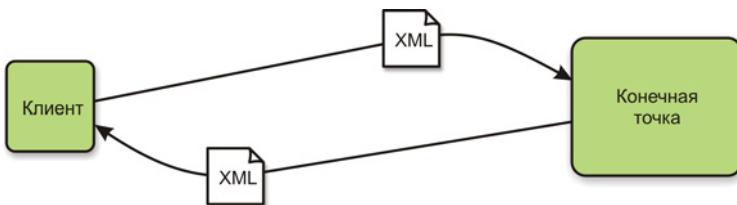


Рис. 15.3. Конечные точки – это реализация веб-службы в Spring-WS.
Конечные точки обрабатывают входящие сообщения и возвращают ответы в формате XML

Как только конечная точка завершит обработку, она вернет ответ – другое XML-сообщение. В данном случае – XML-сообщение `<EvaluateHandResponse>`.

Фреймворк Spring-WS определяет несколько абстрактных классов, перечисленных в табл. 15.2, на основе которых можно создавать конечные точки.

Все абстрактные классы конечных точек, перечисленные в табл. 15.2, в значительной степени похожи друг на друга. Выбор того или другого в большей степени определяется личными предпочтениями и технологией парсинга XML-сообщений (например, SAX, DOM или StAX). Особняком стоит только класс `AbstractMarshallingPayloadEndpoint`, поддерживающий автоматический маршалинг и демаршалинг XML-сообщений в Java-объекты и обратно.

Таблица 15.2. Абстрактные реализации конечных точек в Spring-WS

Абстрактный класс конечной точки в пакете <code>org.springframework.ws.server.endpoint</code>	Описание
<code>AbstractDom4jPayloadEndpoint</code>	Конечная точка, обрабатывающая содержимое сообщений как элементы <code>Element</code> из модели dom4j
<code>AbstractDomPayloadEndpoint</code>	Конечная точка, обрабатывающая содержимое сообщений как элементы <code>Element</code> из модели DOM
<code>AbstractJDomPayloadEndpoint</code>	Конечная точка, обрабатывающая содержимое сообщений как элементы <code>Element</code> из модели JDOM
<code>AbstractMarshallingPayloadEndpoint</code>	Конечная точка, выполняющая демаршалинг содержимого запросов в объекты и маршалинг объектов ответа в XML
<code>AbstractSaxPayloadEndpoint</code>	Конечная точка, обрабатывающая содержимое сообщений с помощью SAX-реализации <code>ContextHandler</code>
<code>AbstractStaxEventPayloadEndpoint</code>	Конечная точка, обрабатывающая содержимое сообщений с помощью StAX-реализации, основанной на событиях
<code>AbstractStaxStreamPayloadEndpoint</code>	Конечная точка, обрабатывающая содержимое сообщений с помощью потоковой StAX-реализации
<code>AbstractXomPayloadEndpoint</code>	Конечная точка, обрабатывающая содержимое сообщений как элементы <code>Element</code> из модели XOM

Класс `AbstractMarshallingPayloadEndpoint` будет рассматриваться немного ниже в этой главе (в разделе 15.3.2). Но сначала посмотрим, как создать конечную точку, обрабатывающую XML-сообщения непосредственно.

15.3.1. Создание конечной точки на основе модели JDOM

Веб-служба, оценивающая комбинацию карт при игре в покер, принимает сообщение `<EvaluateHandRequest>` и возвращает сообщение `<EvaluateHandResponse>`. То есть нам необходимо создать конечную точку веб-службы, обрабатывающую элемент `<EvaluateHandRequest>` и возвращающую элемент `<EvaluateHandResponse>`.

Для этого можно использовать любой из абстрактных классов конечных точек, перечисленных в табл. 15.2, но мы выберем класс `AbstractJDomPayloadEndpoint`. Отчасти выбор был сделан случайно, но лично мне нравится поддержка XPath в JDOM, обеспечивающая простой способ извлечения информации из элементов `Element` в модели JDOM. (За дополнительной информацией о модели JDOM обращайтесь на домашнюю страницу проекта: <http://www.jdom.org>.)

Класс `EvaluateHandJDomEndpoint` (листинг 15.2) наследует класс `AbstractJDomPayloadEndpoint` и предоставляет функциональность, необходимую для обработки сообщения `<EvaluateHandRequest>`.

Листинг 15.2. Конечная точка для обработки сообщений `<EvaluateHandRequest>`

```
package com.springinaction.poker.webservice;
import java.util.Iterator;
import java.util.List;
import org.jdom.Element;
import org.jdom.JDOMException;
import org.jdom.Namespace;
import org.jdom.xpath.XPath;
import org.springframework.beans.factory.InitializingBean;
import org.springframework.ws.server.endpoint.AbstractJDomPayloadEndpoint;
import com.springinaction.poker.Card;
import com.springinaction.poker.Face;
import com.springinaction.poker.PokerHand;
import com.springinaction.poker.PokerHandEvaluator;
import com.springinaction.poker.PokerHandType;
import com.springinaction.poker.Suit;

public class EvaluateHandJDomEndpoint
    extends AbstractJDomPayloadEndpoint
    implements InitializingBean {
    private Namespace namespace;
    private XPath cardsXPath;
```

```
private XPath suitXPath;
private XPath faceXPath;

protected Element invokeInternal(Element element)
    throws Exception {

    Card cards[] = extractCardsFromRequest(element);
    PokerHand pokerHand = new PokerHand();
    pokerHand.setCards(cards);

    PokerHandType handType = pokerHandEvaluator.evaluateHand(pokerHand); // Оценивает комбинацию карт

    return createResponse(handType);
}

private Element createResponse(PokerHandType handType) {// Создает ответ
    Element responseElement =
        new Element("EvaluateHandResponse", namespace);
    responseElement.addContent(
        new Element("handName", namespace).setText(
            handType.toString()));
    return responseElement;
}

private Card[] extractCardsFromRequest(Element element)
    throws JDOMException {
    Card[] cards = new Card[5];

    // Извлечь карты из сообщения
    List cardElements = cardsXPath.selectNodes(element);
    for(int i=0; i < cardElements.size(); i++) {
        Element cardElement = (Element) cardElements.get(i);
        Suit suit = Suit.valueOf(
            suitXPath.valueOf(cardElement));
        Face face = Face.valueOf(
            faceXPath.valueOf(cardElement));
        cards[i] = new Card();
        cards[i].setFace(face);
        cards[i].setSuit(suit);
    }
    return cards;
}

public void afterPropertiesSet() throws Exception {
```

```
namespace = Namespace.getNamespace("poker",
    "http://www.springinaction.com/poker/schemas");

// Подготовить XPath-запросы
cardsXPath =
    XPath.newInstance("//poker:EvaluateHandRequest/poker:card");
cardsXPath.addNamespace(namespace);
faceXPath = XPath.newInstance("poker:face");
faceXPath.addNamespace(namespace);
suitXPath = XPath.newInstance("poker:suit");
suitXPath.addNamespace(namespace);
}

// внедряется
private PokerHandEvaluator pokerHandEvaluator;
public void setPokerHandEvaluator(
    PokerHandEvaluator pokerHandEvaluator) {
    this.pokerHandEvaluator = pokerHandEvaluator;
}
}
```

Метод `invokeInternal()` – это точка входа в данную конечную точку. При вызове ему передается JDOM-объект `Element`, содержащий входящее сообщение, в данном случае `<EvaluateHandRequest>`. Метод `invokeInternal()` передает объект `Element` методу `extractCardsFromRequest()`, который с помощью объектов JDOM XPath извлекает информацию о картах из элемента `<EvaluateHandRequest>`.

После получения массива объектов `Card` метод `invokeInternal()` передает их внедренному объекту `PokerHandEvaluator` для оценки комбинации карт. Объект `PokerHandEvaluator` реализует следующий интерфейс:

```
package com.springinaction.poker;

public interface PokerHandEvaluator {
    PokerHandType evaluateHand(PokerHand hand);
}
```

Фактическая реализация интерфейса `PokerHandEvaluator` не имеет отношения к обсуждению особенностей создания веб-служб с помощью Spring-WS, поэтому я опущу ее (но вы можете найти ее в загружаемых примерах к книге).

Тот факт, что точка входа вызывает метод `evaluateHand()` класса `PokerHandEvaluator`, имеет большое значение. Правильно спроектированная конечная точка Spring-WS не должна выполнять какую-либо прикладную логику. Она должна играть роль лишь передаточного звена между клиентом и внутренним API. Фактическая прикладная логика выполняется реализацией интерфейса `in PokerHandEvaluator`. Подобную организацию мы уже видели в главе 8, где контроллеры Spring MVC играли роль посредников между пользователем и объектами на стороне сервера.

Как только `PokerHandEvaluator` определит значение комбинации карт, метод `invokeInternal()` передаст объект `PokerHandType` методу `createResponse()`, чтобы произвести ответ `<EvaluateHandResponse>`, и после получения объекта `Element` класс `EvaluateHandJDomEndpoint` завершает работу.

`EvaluateHandJDomEndpoint` – отличный пример реализации конечной точки Spring-WS. Но в ней присутствует множество специфических особенностей, имеющих отношение к формату XML. Даже при том, что сообщения, обрабатываемые конечными точками Spring-WS, имеют формат XML, нет никаких причин, почему конечная точка должна быть реализована с учетом этой особенности. Посмотрим далее, как с помощью механизма маршалинга избавиться от операций, связанных с парсингом XML-сообщений.

15.3.2. Маршалинг содержимого сообщений

Как отмечалось выше, класс `AbstractMarshallingPayloadEndpoint` несколько отличается от других абстрактных классов конечных точек в Spring-WS. Вместо XML-элемента конечная точка `AbstractMarshallingPayloadEndpoint` получает Java-объект для обработки.

Фактически, как показано на рис. 15.4, конечная точка действует в паре с демаршалером, преобразующим XML-сообщения в POJO. Завершив обработку, конечная точка просто возвращает POJO, а маршалер превращает его в XML-сообщение, которое затем передается клиенту. Такой подход позволяет существенно упростить реализацию конечной точки, так как ей не приходится заниматься обработкой разметки XML.

Например, рассмотрим листинг 15.3, где приводится новая реализация конечной точки – класс `EvaluateHandMarshallingEndpoint`, наследующий класс `AbstractMarshallingPayloadEndpoint`.

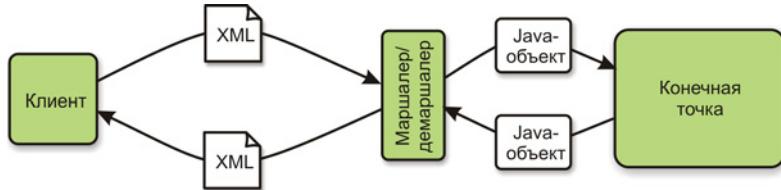


Рис. 15.4. Конечная точка снабжается промежуточным объектом, занимающимся промежуточными преобразованиями XML-сообщений, благодаря чему конечной точке приходится работать только с простыми Java-объектами

Листинг 15.3. Конечная точка, обрабатывающая сообщения <EvaluateHandRequest>

```

package com.springinaction.poker.webservice;
import org.springframework.ws.server.endpoint.AbstractMarshallingPayloadEndpoint;
import com.springinaction.poker.PokerHand;
import com.springinaction.poker.PokerHandEvaluator;
import com.springinaction.poker.PokerHandType;

public class EvaluateHandMarshallingEndpoint
    extends AbstractMarshallingPayloadEndpoint {

    // Передает в конечную точку реализацию EvaluateHandRequest
    protected Object invokeInternal(Object object)
        throws Exception {
        EvaluateHandRequest request =
            (EvaluateHandRequest) object;

        PokerHand pokerHand = new PokerHand();
        pokerHand.setCards(request.getHand());

        PokerHandType pokerHandType =           // Оценивает комбинацию карт
            pokerHandEvaluator.evaluateHand(pokerHand);

        return new EvaluateHandResponse(pokerHandType);
    }

    // внедряется
    private PokerHandEvaluator pokerHandEvaluator;
    public void setPokerHandEvaluator(
        PokerHandEvaluator pokerHandEvaluator) {
        this.pokerHandEvaluator = pokerHandEvaluator;
    }
}

```

Первое, на что следует обратить внимание, – класс EvaluateHandMarshallingEndpoint получился намного короче класса EvaluateHandJDomEndpoint. Это обусловлено отсутствием в классе EvaluateHandMarshallingEndpoint операций парсинга разметки XML, которые были необходимы в классе EvaluateHandJDomEndpoint.

В этой версии методу invokeInternal() передается объект Object. В данном случае – объект класса EvaluateHandRequest:

```
package com.springinaction.poker.webservice;
import com.springinaction.poker.Card;

public class EvaluateHandRequest {
    private Card[] hand;

    public EvaluateHandRequest() {}

    public Card[] getHand() {
        return hand;
    }

    public void setHand(Card[] cards) {
        this.hand = cards;
    }
}
```

С другой стороны, метод invokeInternal() возвращает объект EvaluateHandResponse. Ниже приводится определение класса EvaluateHandResponse:

```
package com.springinaction.poker.webservice;
import com.springinaction.poker.PokerHandType;

public class EvaluateHandResponse {
    private PokerHandType pokerHand;

    public EvaluateHandResponse() {
        this(PokerHandType.NONE);
    }

    public EvaluateHandResponse(PokerHandType pokerHand) {
        this.pokerHand = pokerHand;
    }

    public PokerHandType getPokerHand() {
```

```
    return this.pokerHand;
}

public void setPokerHand(PokerHandType pokerHand) {
    this.pokerHand = pokerHand;
}
}
```

Но как входящее XML-сообщение `<EvaluateHandRequest>` преобразуется в объект `EvaluateHandRequest?` И как возвращаемый объект `EvaluateHandResponse` превращается в сообщение `<EvaluateHandResponse>`, отправляемое клиенту?

От наших глаз укрылось одно обстоятельство: класс `AbstractMarshallingPayloadEndpoint` имеет ссылку на объект, выполняющий маршалинг разметки XML. Когда класс конечной точки получает XML-сообщение, перед вызовом `invokeInternal()` он с помощью демаршалера превращает XML-сообщение в объект. Затем, когда `invokeInternal()` завершится, маршалер превратит Java-объект в XML-сообщение.

Значительную часть фреймворка Spring-WS составляет реализация механизмов отображения объектов в XML и обратно (Object-XML Mapping, OXM). Фреймворк Spring-WS включает несколько реализаций OXM, включая:

- ❑ JAXB (версии 1 и 2);
- ❑ Castor XML;
- ❑ JiBX;
- ❑ XMLBeans;
- ❑ XStream.

Кого-то может заинтересовать вопрос, какой из механизмов OXM я выбрал для реализации `EvaluateHandMarshallingEndpoint`. Я отвечу на этот вопрос, но не сейчас. Важно отметить, что `EvaluateHandMarshallingEndpoint` понятия не имеет, откуда берется объект `Object`, передаваемый методу `invokeInternal()`. Фактически объект `Object` вообще может быть создан не на основе разметки XML.

Это обстоятельство подчеркивает ключевое преимущество конечной точки с маршалингом. Поскольку теперь класс `EvaluateHandMarshallingEndpoint` получает простой объект, он легко может быть подвергнут модульному тестированию, подобно любому другому POJO. Испытательный тест может просто передавать конечной точке объект `EvaluateHandRequest` и выполнять проверки возвращаемого объекта `EvaluateHandResponse`.

Теперь, когда конечная точка службы готова, можно приступать к ее связыванию.

15.4. Связываем все вместе

Мы наконец достигли заключительной стадии разработки службы Spring-WS. Теперь необходимо сконфигурировать контекст приложения Spring, внедрив в него компонент конечной точки и некоторые компоненты, образующие инфраструктуру Spring-WS.

Фреймворк Spring-WS основан на фреймворке Spring MVC (который описывался в главе 8). В Spring MVC все запросы попадают в специальный сервлет `DispatcherServlet`, который передает запросы классам контроллеров для обработки. Аналогичным образом действует и фреймворк Spring-WS: запросы сначала попадают в сервлет `MessageDispatcherServlet`, подкласс класса `DispatcherServlet`, который передает SOAP-запросы конечным точкам Spring-WS¹.

`MessageDispatcherServlet` – это удивительно простой сервлет, и его можно настроить в файле `web.xml` веб-приложения с помощью элементов `<servlet>` и `<servletmapping>`, как показано ниже:

```
<servlet>
    <servlet-name>poker</servlet-name>
    <servlet-class>org.springframework.ws.transport.http.
        → MessageDispatcherServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>poker</servlet-name>
    <url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

К конфигурации сервлета `MessageDispatcherServlet` мы еще вернемся чуть ниже, а пока немного отвлечемся.

¹ Считается, что приставка «веб» в слове «веб-служба» предполагает работу всех веб-служб по протоколу HTTP. Но это не так. Фреймворк Spring-WS поддерживает создание веб-служб, действующих по протоколам JMS и электронной почты, а также на основе низкоуровневого протокола TCP/IP. Однако, так как большинство веб-служб фактически действуют по протоколу HTTP, именно этот протокол и будет подразумеваться в описываемой конфигурации.

MessageDispatcherServlet – единственный входной контроллер (front controller) в Spring-WS. Однако фреймворк Spring-WS содержит еще ряд компонентов, которые также необходимо внедрить в контекст приложения Spring. Познакомимся с этими компонентами поближе.

15.4.1. Spring-WS: общая картина

На протяжении следующих нескольких страниц мы займемся настройкой дополнительных компонентов в контексте Spring. Но, прежде чем углубиться в XML, вероятно, стоит получить общее представление о том, что мы собираемся делать. На рис. 15.5 показаны компоненты, которые нам предстоит определить, и как они связаны друг с другом.

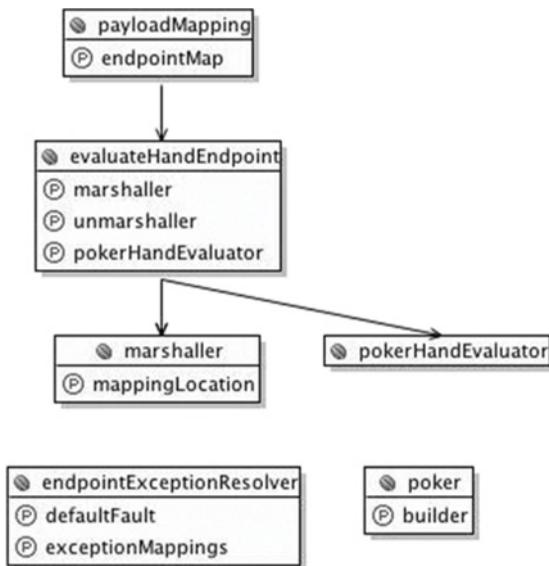


Рис. 15.5. Конфигурация службы на основе Spring-WS
состоит из нескольких компонентов, включая отображения, конечные точки, маршалеры и другие вспомогательные компоненты

На рис. 15.5 показаны компоненты службы оценки комбинации карт при игре в покер, которые предстоит настроить, и взаимосвязи между ними. Но для чего нужны эти компоненты и что они делают? Всего необходимо сконфигурировать шесть компонентов.

- ❑ payloadMapping – отображает входящие XML-сообщения в соответствующую конечную точку. В данном случае используется отображение, определяющее конечную точку на основе корневого элемента входящего XML-сообщения.
- ❑ evaluateHandEndpoint – конечная точка, обрабатывающая входящие XML-сообщения для службы оценки комбинации карт при игре в покер.
- ❑ marshaller – в конечной точке evaluateHandEndpoint можно было бы реализовать обработку входящих XML-сообщений с применением модели DOM или JDOM, или даже с помощью парсера SAX. Однако было решено упростить ее и переложить преобразование XML-сообщений в Java-объекты и обратно на компонент marshaller.
- ❑ pokerHandEvaluator – это POJO, реализующий обработку комбинации карт. Конечная точка evaluateHandEndpoint будет использовать этот компонент для получения результата.
- ❑ endpointExceptionResolver – это компонент Spring-WS, который автоматически преобразует любые исключения Java, возникшие в процессе обработки, в соответствующие ошибки SOAP.
- ❑ poker – хотя этого и не следует из имени данного компонента, он служит для передачи WSDL-определения веб-службы клиенту. Это может быть файл WSDL, созданный вручную или сгенерированный автоматически, на основе описания XML-сообщений на языке XML Schema.

Теперь, когда стало известно, куда двигаться дальше, погрузимся в настройку Spring-WS, начав с отображения сообщений в конечные точки.

15.4.2. Отображение сообщений в конечные точки

Как после получения сообщения от клиента сервер MessageDispatcherServlet определит, какая конечная точка должна его обрабатывать? Даже при том, что в данном примере создается единственная конечная точка (оценивающая комбинацию карт), сервер MessageDispatcherServlet поддерживает работу с несколькими конечными точками. Нам необходим некоторый способ, который позволит отображать входящие сообщения в конечные точки, обрабатывающие их.

В главе 8 было показано, как сервер DispatcherServlet из Spring MVC отображает запросы браузера в контроллеры Spring MVC. Ана-

логичным образом действует и сервлет MessageDispatcherServlet, выполняющий отображение входящих XML-сообщений в конечные точки.

Для нужд службы оценки комбинации карт мы будем использовать компонент PayloadRootQNameEndpointMapping из фреймворка Spring-WS, который настраивается, как показано ниже:

```
<bean id="payloadMapping"
      class="org.springframework.ws.server.endpoint.mapping.
           ↳ PayloadRootQNameEndpointMapping">
    <property name="endpointMap">
      <map>
        <entry key=
               "{http://www.springinaction.com/poker/schemas}EvaluateHandRequest"
               value-ref="evaluateHandEndpoint" />
      </map>
    </property>
</bean>
```

Компонент PayloadRootQNameEndpointMapping отображает входящие SOAP-сообщения в конечные точки, исследуя полное квалифицированное имя (QName) содержимого сообщения и определяя соответствующую ему конечную точку в списке (свойство endpointMap).

В нашем примере корневым элементом сообщения является <EvaluateHandRequest>. С идентификатором URI пространства имен <http://www.springinaction.com/poker/schemas> получается полное имя сообщения {<http://www.springinaction.com/poker/schemas>}EvaluateHandRequest. Это имя отображается в компонент с идентификатором evaluateHandEndpoint, который является реализацией конечной точки, созданной в разделе 15.3.2.

15.4.3. Настройка конечной точки службы

Мы, наконец, подошли к внедрению конечной точки, которая будет обрабатывать сообщения. Если вы предпочтете использовать реализацию конечной точки, основанной на модели JDOM, тогда она должна настраиваться, как показано ниже:

```
<bean id="evaluateHandEndpoint"
      class="com.springinaction.poker.webservice.EvaluateHandJDomEndpoint">
    <property name="pokerHandEvaluator"
              ref="pokerHandEvaluator" />
</bean>
```

Единственное, куда следует выполнить внедрение, – свойство `pokerHandEvaluator`. Напомню, что сама конечная точка `EvaluateHandJDomEndpoint` сама не реализует оценку комбинации карт – она делегирует эту грязную работу реализации интерфейса `PokerHandEvaluator`. То есть необходимо настроить компонент `pokerHandEvaluator`, как показано ниже:

```
<bean id="pokerHandEvaluator"
      class="com.springinaction.poker.PokerHandEvaluatorImpl"/>
```

Если вы решили не использовать реализацию конечной точки, основанной на модели JDOM, а отдали предпочтение реализации, использующей `EvaluateHandMarshallingEndpoint`, необходимо выполнить дополнительные настройки:

```
<bean id="evaluateHandEndpoint"
      class="com.springinaction.poker.webservice.
           ↳ EvaluateHandMarshallingEndpoint">
    <property name="marshaller" ref="marshaller" />
    <property name="unmarshaller" ref="marshaller" />
    <property name="pokerHandEvaluator"
              ref="pokerHandEvaluator" />
</bean>
```

Повторюсь, что в обеих версиях в свойство `pokerHandEvaluator` необходимо внедрить ссылку на реализацию `PokerHandEvaluatorImpl`. А в версии конечной точки с поддержкой маршалинга необходимо дополнительно определить свойства `marshaller` и `unmarshaller`. В примере выше обоим свойствам присвоена ссылка на один и тот же компонент `marshaller`, настройка которого описывается далее.

15.4.4. Настройка маршалера сообщений

Преобразование объектов в формат XML и обратно выполняется с помощью механизма отображения объектов в XML (Object-XML Mapping, OXM). Фреймворк Spring-OXM является составной частью фреймворка Spring-WS и реализует уровень абстракции на основе нескольких популярных OXM-решений, включая JAXB и Castor XML.

Центральное положение в Spring-OXM занимают интерфейсы `Marshaller` и `Unmarshaller`. Реализации интерфейса `Marshaller`, как ожидается, должны из Java-объектов генерировать XML-элементы.

Реализации интерфейса `Unmarshaller`, напротив, используются для конструирования Java-объектов из XML-элементов.

При обработке сообщений класс `AbstractMarshallingPayloadEndpoint` использует маршалеры и демаршалеры из Spring-OXM. Когда `AbstractMarshallingPayloadEndpoint` принимает сообщение, оно передается реализации интерфейса `Unmarshaller` для преобразования XML-сообщения в объект, который затем передается методу `invokeInternal()`. Затем, когда `invokeInternal()` завершит работу, возвращаемый им объект передается реализации интерфейса `Marshaller` для преобразования в XML-сообщение, которое будет отправлено клиенту.

К счастью, вам не придется создавать собственные реализации интерфейсов `Marshaller` и `Unmarshaller`. Фреймворк Spring-OXM уже содержит несколько таких реализаций, которые перечислены в табл. 15.3.

Таблица 15.3. Маршалеры, преобразующие объекты в формат XML и обратно. Фреймворк Spring-OXM предоставляет несколько реализаций маршалеров, которые можно использовать совместно с фреймворком Spring-WS

ОХМ-решение	Маршалер в Spring-OXM
Castor XML	<code>org.springframework.oxm.castor.CastorMarshaller</code>
JAXB v1	<code>org.springframework.oxm.jaxb.JAXB1Marshaller</code>
JAXB v2	<code>org.springframework.oxm.jaxb.JAXB2Marshaller</code>
JiBX	<code>org.springframework.oxm.jibx.JiBXMarshaller</code>
XMLBeans	<code>org.springframework.oxm.xmlbeans.XmlBeansMarshaller</code>
XStream	<code>org.springframework.oxm.xstream.XStreamMarshaller</code>

В табл. 15.3 перечислены только классы маршалеров. Но это не ошибка. Все классы маршалеров, перечисленные в табл. 15.3, реализуют оба интерфейса, `Marshaller` и `Unmarshaller`, обеспечивая универсальные ОХМ-решения маршалинга.

Выбор того или иного ОХМ-решения – в значительной степени дело вкуса. Каждое решение, предлагаемое фреймворком Spring-WS, имеет свои достоинства и недостатки. Однако решение XStream имеет ограниченную поддержку пространств имен XML, необходимых в определениях типов для веб-служб. То есть решение XStream с успехом может использоваться для сериализации объектов в формат XML и обратно вообще, но его не следует применять при разработке веб-служб.

Для нужд службы оценки комбинации карт было выбрано решение Castor XML. Таким образом, нам необходимо настроить компонент CastorMarshaller в контексте Spring:

```
<bean id="marshaller"
      class="org.springframework.oxm.castor.CastorMarshaller">
    <property name="mappingLocation"
              value="classpath:mapping.xml" />
</bean>
```

Решение Castor XML можно использовать для выполнения простого XML-маршалинга без дополнительных настроек. Но нам требуется немного больше, чем может дать решение Castor XML с настройками по умолчанию. Следовательно, компонент CastorMarshaller необходимо настроить на использование файла отображения. Свойство mappingLocation определяет местоположение файла отображения в формате Castor XML. В примере выше посредством свойства mappingLocation мы указали, что поиск файла отображения с именем mapping.xml должен выполняться в корневом каталоге библиотеки классов приложения.

Содержимое файла mapping.xml приводится в листинге 15.4.

Листинг 15.4. Файл отображения в формате Castor XML с определениями типов для службы оценки комбинации карт при игре в покер

```
<?xml version="1.0"?>
<!DOCTYPE mapping PUBLIC
  "-//EXOLAB/Castor Object Mapping DTD Version 1.0//EN"
  "http://castor.exolab.org/mapping.dtd">

<!-- Отображает &lt;EvaluateHandRequest&gt; в EvaluateHandRequest --&gt;
&lt;mapping xmlns="http://castor.exolab.org/"&gt;
&lt;class name="com.springinaction.poker.webservice.EvaluateHandRequest"&gt;
  &lt;map-to xml="EvaluateHandRequest" /&gt;

  <!-- Отображает &lt;hand&gt; в массив объектов Card --&gt;
  &lt;field name="hand" collection="array"
        type="com.springinaction.poker.Card"
        required="true"&gt;
    &lt;bind-xml name="card" node="element" /&gt;
  &lt;/field&gt;
&lt;/class&gt;

<!-- Отображает &lt;card&gt; в Card и &lt;suit&gt; в Suit--&gt;</pre>
```

```
<class name="com.springinaction.poker.Card">
    <map-to xml="card" />
    <field name="suit" type="com.springinaction.poker.Suit"
        required="true">
        <bind-xml name="suit" node="element" />
    </field>

    <!-- Отображает <face> в Face -->
    <field name="face" type="com.springinaction.poker.Face"
        required="true">
        <bind-xml name="face" node="element" />
    </field>
</class>

<!-- Отображает <EvaluateHandResponse> в EvaluateHandResponse -->
<class name="com.springinaction.poker.webservice.EvaluateHandResponse">
    <map-to xml="EvaluateHandResponse"
        ns-uri="http://www.springinaction.com/poker/schemas"
        ns-prefix="tns" />
    <field name="pokerHand"
        type="com.springinaction.poker.PokerHandType"
        required="true">
        <bind-xml name="tns:handName" node="element"
            QName-prefix="tns"
            xmlns:tns="http://www.springinaction.com/poker/schemas"/>
    </field>
</class>
</mapping>
```

Теперь у нас имеются настроенные компоненты отображения сообщений в конечные точки, реализации конечной точки и маршаллинга XML. На данный момент реализация веб-службы оценки комбинации карт при игре в покер практически закончена. Мы могли бы развернуть приложение и закончить на этом. Но у нас осталась еще пара компонентов, которые сделают веб-службу более полной. Посмотрим, как повысить надежность веб-службы, добавив компонент, отображающий исключения Java в ошибки SOAP.

15.4.5. Обработка исключений в конечной точке

Программный код не всегда действует, как хотелось бы. Что произойдет, если сообщение не сможет быть преобразовано в Java-объект? Что, если сообщение вообще не будет являться допустимым

XML-документом? Что необходимо сделать, если конечная точка или одна из ее зависимостей возбудит исключение?

Если в процессе обработки будет возбуждено исключение, клиенту должно быть отправлено соответствующее сообщение об ошибке. К сожалению, протокол SOAP ничего не знает об исключениях, возникающих в программном коде Java. Веб-службы, действующие по протоколу SOAP, возвращают сообщения об ошибках в виде ошибок SOAP. Поэтому нам необходим некоторый механизм, преобразующий любые исключения Java, которые могут быть возбуждены веб-службой или фреймворком Spring-WS, в ошибки SOAP.

Для этого в Spring-WS имеется компонент `SoapFaultMappingExceptionResolver`. Как показано на рис. 15.6, `SoapFaultMappingExceptionResolver` обрабатывает любые неперехваченные исключения, которые могут возникнуть в ходе обработки сообщения, и воспроизводит соответствующую ошибку SOAP для передачи клиенту.

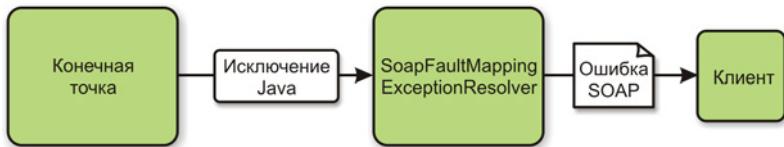


Рис. 15.6. Компонент `SoapFaultMappingExceptionResolver` отображает любые исключения Java в ошибки SOAP для передачи клиенту

Настройки компонента `SoapFaultMappingExceptionResolver` для нашей службы приводятся ниже:

```

<bean id="endpointExceptionResolver"
      class="org.springframework.ws.soap.server.endpoint.
           SoapFaultMappingExceptionResolver">
    <property name="exceptionMappings">
      <props>
        <prop key="org.springframework.oxm.UnmarshallingFailureException">
          SENDER, Invalid message received</prop>
        <prop key="org.springframework.oxm.ValidationFailureException">
          SENDER, Invalid message received</prop>
      </props>
    </property>
    <property name="defaultFault">
      value="RECEIVER, Server error" />
  </bean>
  
```

Свойство exceptionMappings определяет перечень исключений и соответствующих им ошибок SOAP. Атрибут key каждого элемента <ргор> определяет исключение Java, которое требуется преобразовать в ошибку SOAP. Содержимое элемента <ргор> определяет два значения – тип ошибки, которая должна быть создана, и строка с описанием.

Всего в протоколе SOAP имеются два типа ошибок: ошибка отправителя и ошибка получателя. Ошибки отправителя обычно указывают на проблемы, возникшие на стороне клиента (то есть отправителя). Ошибки получателя указывают, что веб-служба (то есть получатель) благополучно приняла сообщение от клиента, но столкнулась с некоторыми проблемами при его обработке.

Например, если служба примет XML-сообщение, которое не может быть преобразовано в Java-объект, маршалер возбудит исключение org.springframework.oxm.UnmarshallingFailureException. Поскольку XML-сообщения создаются на стороне отправителя, это ошибка отправителя. Что касается строки с описанием, это простая строка с текстом «Invalid message received» («Принято недопустимое сообщение»), описывающая характер проблемы. Исключение org.springframework.oxm.ValidationFailureException обрабатывается аналогично.

Любые исключения, которые явно не определены в свойстве exceptionMappings, будут обрабатываться в соответствии с настройками свойства defaultFault. В данном случае предполагается, что если было возбуждено исключение, не соответствующее ни одному из указанных явно, следовательно, проблема возникла на принимающей стороне. То есть это ошибка получателя, которой соответствует описание «Server error» («Ошибка на стороне сервера»).

15.4.6. Создание WSDL-файлов

Наконец, я собираюсь выполнить свое обещание и показать, откуда берется WSDL-файл с определением веб-службы оценки комбинации карт при игре в покер. В разделе 15.2.1 мы уже определили формат представления данных в виде схемы на языке XML Schema, в файле PokerTypes.xsd. Прежде чем двинуться дальше, вернитесь к листингу 15.1, чтобы освежить в памяти, как выглядит определение формата представления данных.

Особое внимание обратите на имена, которые я выбрал для XML-элементов, определяющих типы сообщений для веб-службы: EvaluateHandRequest и EvaluateHandResponse. Эти имена были выбраны

не случайно. Они выбирались целенаправленно, с учетом поддержки в Spring-WS принципа преимущества соглашений перед настройками, которая будет автоматически создавать WSDL-файл для службы оценки комбинации карт.

Чтобы задействовать эту поддержку, необходимо настроить специальный компонент `DynamicWSDL11Definition`. `DynamicWSDL11Definition`, который используется сервлетом `MessageDispatcherServlet` для создания WSDL-определения из схемы на языке XML Schema. Это очень удобно, так как у нас уже имеется схема XML Schema, определяющая формат представления данных. Ниже показано, как я настроил компонент `DynamicWSDL11Definition` в контексте Spring:

```
<bean id="poker"
      class="org.springframework.ws.wsdl.wsdl11.DynamicWSDL11Definition">
    <property name="builder">
      <bean class="org.springframework.ws.wsdl.wsdl11.builder.
                   ➤ XsdBasedSoap11WSDL4jDefinitionBuilder">
        <property name="schema" value="/PokerTypes.xsd"/>
        <property name="portTypeName" value="Poker"/>
        <property name="locationUri"
                  value="http://localhost:8080/Poker-WS/services"/>
      </bean>
    </property>
</bean>
```

Компонент `DynamicWSDL11Definition` читает описание схемы на языке XML Schema из файла `PokerTypes.xsd`, как определено свойством `schema`. Он отыскивает в файле все определения элементов, оканчивающихся словом `Request` или `Response`. И исходя из предположения, что эти окончания соответствуют сообщениям, посылаемых операциям службы или возвращаемых ими, создает соответствующие элементы `<wsdl:operation>` в WSDL-определении, как показано на рис. 15.7.

Например, обрабатывая файл `PokerTypes.xsd`, компонент `DynamicWSDL11Definition` предполагает, что элементы `EvaluateHandRequest` и `EvaluateHandResponse` описывают входящее и исходящее сообщения для операции с именем `EvaluateHand`. В результате он воспроизводит следующее определение WSDL:

```
<wsdl:portType name="Poker">
  <wsdl:operation name="EvaluateHand">
    <wsdl:input message="schema:EvaluateHandRequest"
                name="EvaluateHandRequest">
```

```

</wsdl:input>
<wsdl:output message="schema:EvaluateHandResponse"
    name="EvaluateHandResponse">
</wsdl:output>
</wsdl:operation>
</wsdl:portType>

```

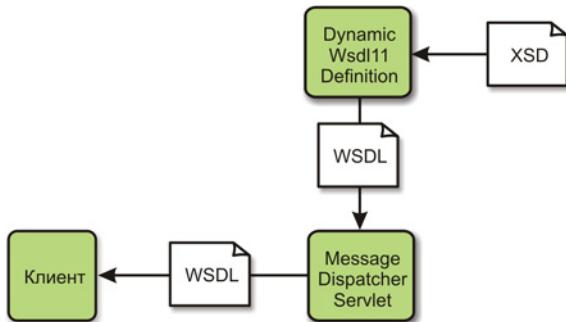


Рис. 15.7. Компонент DynamicWsdl11Definition автоматически воспроизводит WSDL-определение для веб-службы, опираясь на описание схемы на языке XML Schema, определяющей сообщения, которыми клиенты обмениваются со службой

Обратите внимание, что компонент DynamicWsdl11Definition поместил элемент `<wsdl:operation>` с именем `EvaluateHand` в элемент `<wsdl:portType>` с именем `Poker`. Имя для элемента `<wsdl:portType>` было взято из свойства `portTypeName`.

Последним в компоненте DynamicWsdl11Definition мы настроили свойство `locationUri`. Это свойство сообщает клиенту адрес, где находится служба. Диаграмма на рис. 15.8 изображает структуру адреса URL, который присваивается свойству `locationUri`.



Рис. 15.8. Структура адреса URL в свойстве `locationUri`

В данном случае предполагается, что служба будет действовать на локальном компьютере, поэтому, если вы соберетесь опробовать ее на удаленном компьютере, вам придется изменить URL. Обратите внимание, что URL оканчивается строкой /services в соответствии с настройками в элементе <servlet-mapping> для компонента MessageDispatcherServlet.

Коль скоро мы заговорили об элементе <servlet-mapping>, нам также необходимо добавить в файл web.xml новый элемент <servlet-mapping>, чтобы компонент MessageDispatcherServlet мог генерировать определение WSDL. Определение нового элемента <servlet-mapping> показано ниже:

```
<servlet-mapping>
    <servlet-name>poker</servlet-name>
    <url-pattern>*.wsdl</url-pattern>
</servlet-mapping>
```

Теперь сервлет MessageDispatcherServlet сможет автоматически генерировать (с помощью DynamicWsdl11Definition) определение WSDL службы оценки комбинации карт при игре в покер. Единственный вопрос, который пока остается без ответа, – где искать сгенерированное определение WSDL.

Сгенерированное определение WSDL можно найти по адресу <http://localhost:8080/Poker-WS/poker.wsdl>. Как я это узнал? Я знаю, что контроллер MessageDispatcherServlet отображается в шаблон *.wsdl, поэтому при обращении по любому адресу, соответствующему этому шаблону, будет создано определение WSDL. Но откуда он знает, что определение WSDL для нашей службы должно быть сгенерировано в виде файла с именем poker.wsdl?

Ответ на этот вопрос находится в последнем пункте соглашения, которому следует MessageDispatcherServlet. Обратите внимание, что я объявил компонент DynamicWsdl11Definition с идентификатором poker. Когда компонент MessageDispatcherServlet получит запрос для URL /poker.wsdl, он будет искать в контексте Spring компонент с именем poker, создающий определение WSDL. В данном случае он найдет компонент DynamicWsdl11Definition.

Использование предопределенного WSDL-определения

Компонент DynamicWsdl11Definition с успехом можно использовать в самых разных ситуациях, так как он избавляет от необходимости

писать WSDL-определения вручную. Но иногда может потребоваться более полный контроль над WSDL-определением службы. В таких ситуациях бывает желательно вручную создать WSDL-файл и затем внедрить его в контекст Spring с помощью `SimpleWsdl11Definition`:

```
<bean id="poker"
      class="org.springframework.ws.wsdl.wsdl11.SimpleWsdl11Definition">
    <property name="wsdl" value="/PokerService.wsdl"/>
</bean>
```

Компонент `SimpleWsdl11Definition` не генерирует WSDL-определение автоматически (рис. 15.9), он просто возвращает WSDL-файл, имя которого указывается в свойстве `wsdl`.

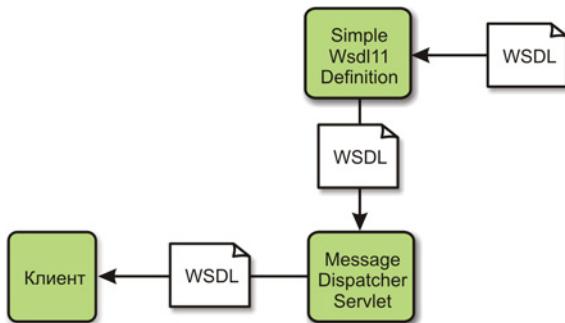


Рис. 15.9. Компонент `SimpleWsdl11Definition` просто возвращает предопределенный WSDL-файл посредством `MessageDispatcherServlet`. При необходимости компонент `MessageDispatcherServlet` можно настроить так, чтобы он изменял адрес службы в соответствии с настройками

Единственная проблема при использовании предопределенного WSDL-файла (кроме усилий, необходимых на его создание) – в том, что он определен статически. Это создает сложности в той части WSDL-определения, где указывается местоположение службы. Например, взгляните на следующий (статический) фрагмент WSDL:

```
<wsdl:service name="PokerService">
  <wsdl:port binding="tns:PokerBinding" name="PokerPort">
    <wsdlsoap:address
      location="http://localhost:8080/Poker-WS/services"/>
  </wsdl:port>
</wsdl:service>
```

Здесь определяется, что служба будет доступна по адресу <http://localhost:8080/Poker-WS/services>. Что, в принципе, вполне годится для этапа разработки, не подходит при развертывании на другом сервере. Можно было бы вручную изменять WSDL-файл при каждом развертывании службы на другом сервере, но это слишком утомительно и есть риск допустить ошибку.

Но компонент `MessageDispatcherServlet` знает, где он разворачивается, и знает, какой адрес URL будет использоваться для запросов к нему. Поэтому вместо редактирования WSDL-файла при развертывании службы на другом сервере почему бы не позволить компоненту `MessageDispatcherServlet` переопределить адрес самостоятельно?

Для этого достаточно добавить параметр `<init-param>` с именем `transformWsdlLocations` и значением `true`, а обо всем остальном позаботится компонент `MessageDispatcherServlet`:

```
<servlet>
    <servlet-name>poker</servlet-name>
    <servlet-class>org.springframework.ws.transport.http.
        ↗ MessageDispatcherServlet</servlet-class>
    <init-param>
        <param-name>transformWsdlLocations</param-name>
        <param-value>true</param-value>
    </init-param>
</servlet>
```

Обнаружив параметр `transformWsdlLocations` со значением `true`, компонент `MessageDispatcherServlet` перезапишет адрес службы в WSDL-файле, возвращаемом компонентом `SimpleWsdl11Definition`, в соответствии с адресом URL запроса.

15.4.7. Развёртывание службы

Итак, мы создали определение службы, реализовали конечную точку и настроили все необходимые компоненты фреймворка Spring-WS. Сейчас все готово к упаковке и развертыванию веб-службы. Поскольку для управления проектом я выбрал инструмент Maven 2, создание WAR-файла для последующего развертывания сводится к выполнению простой команды:

```
% mvn package deploy
```

Как только утилита mvn завершит работу, в целевом каталоге появится файл Poker-WS.war, который можно развернуть на большинстве серверов веб-приложений.

Пример применения фреймворка Spring-WS для создания веб-службы продемонстрировал лишь половину его возможностей. Вторая его половина включает клиентский API, реализующий ту же самую парадигму, основанную на обмене сообщениями. Посмотрим далее, как с помощью Spring-WS создать клиента, использующего службу оценки комбинации карт при игре в покер.

15.5. Использование веб-служб Spring-WS

В главе 11 было показано, как можно использовать компонент JaxWsPortProxyFactoryBean для создания клиентов, взаимодействующих с удаленными веб-службами. Но там удаленные веб-службы интерпретировались как удаленные объекты, чьи методы можно вызывать локально. В этой же главе мы говорим о веб-службах, основанных на обмене сообщениями, когда клиент отправляет веб-службе и принимает в ответ XML-сообщения. Иная парадигма на стороне службы требует использования иной парадигмы на стороне клиента. Для этой цели служит класс WebServiceTemplate из фреймворка Spring-WS.

WebServiceTemplate – это основа клиентского API в Spring-WS. Как показано на рис. 15.10, для реализации отправки и приема XML-сообщений он использует шаблон проектирования «Шаблон» (Template). Мы уже встречались с этим шаблоном проектирования в главе 6, когда знакомились с особенностями реализации уровня абстракции доступа к данным в Spring.

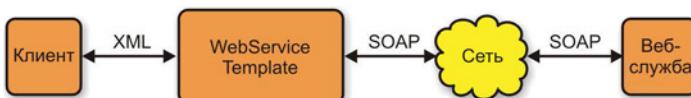


Рис. 15.10. WebServiceTemplate – основной класс клиентского API в Spring-WS.
Он реализует отправку и прием XML-сообщений на стороне клиента

По мере знакомства с клиентским API фреймворка Spring-WS вы обнаружите, что он очень напоминает API доступа к данным.

Для демонстрации особенностей WebServiceTemplate создадим несколько разных реализаций интерфейса PokerClient, который определяется, как показано ниже:

```
package com.springinaction.ws.client;
import java.io.IOException;
import com.springinaction.poker.Card;
import com.springinaction.poker.PokerHandType;

public interface PokerClient {
    PokerHandType evaluateHand(Card[] cards)
        throws IOException;
}
```

В каждой реализации будет демонстрироваться отдельный способ использования класса WebServiceTemplate для отправки сообщений веб-службе оценки комбинации карт при игре в покер.

Но не будем забегать вперед. Сначала настроим компонент WebServiceTemplate в контексте Spring.

15.5.1. Работа с шаблонами веб-служб

Как уже упоминалось, WebServiceTemplate – это основной класс клиентского API в Spring-WS. Отправка сообщения веб-службе включает в себя оформление пакетов SOAP и требует массу шаблонного программного кода, практически одинакового для всех клиентов веб-служб. Реализуя отправку сообщений в клиенте на основе фреймворка Spring-WS, вы определенно пожелаете опереться на класс WebServiceTemplate, выполняющий все рутинные операции, чтобы все свое внимание сосредоточить на прикладной логике.

Настройка компонента WebServiceTemplate в Spring выполняется достаточно просто, как показано ниже:

```
<bean id="webServiceTemplate"
      class="org.springframework.ws.client.core.WebServiceTemplate">
    <property name="messageFactory">
      <bean class="org.springframework.ws.soap.saaj.
        SaajSoapMessageFactory"/>
    </property>
    <property name="messageSender" ref="messageSender"/>
</bean>
```

Компоненту `WebServiceTemplate` необходимо знать, как конструировать сообщения, отправляемые веб-службе, и как отправлять их. Задачу конструирования сообщения берет на себя объект, внедренный в свойство `messageFactory`. Он должен реализовать интерфейс `WebServiceMessageFactory`, объявленный в Spring-WS. К счастью, вам не придется отвлекаться на реализацию интерфейса `WebServiceMessageFactory`, так как Spring-WS уже содержит три такие реализации (перечислены в табл. 15.4).

Таблица 15.4. При создании сообщений, отправляемых веб-службе, компонент `WebServiceTemplate` опирается на специализированный фабричный объект. Фреймворк Spring-WS содержит три реализации фабрики сообщений

Фабрика сообщений	Описание
<code>AxiomSoapMessageFactory</code>	Конструирует SOAP-сообщения, используя модель AXIOM Object Model (AXIOM). Основан на потоковом StAX API. Обладает высокой эффективностью и с успехом может использоваться для обработки больших сообщений
<code>DomPoxMessageFactory</code>	Конструирует сообщения в простом формате XML (Plain Old XML, POX) с использованием модели DOM. Эта фабрика сообщений может использоваться, когда ни клиент, ни веб-служба не поддерживают протокол SOAP
<code>SaajSoapMessageFactory</code>	Конструирует SOAP-сообщения, используя API вложений для Java (Attachments API for Java, SAAJ). Поскольку SAAJ использует модель DOM, при создании больших сообщений могут расходоваться большие объемы памяти. Если производительность приложения имеет большое значение, лучше использовать фабрику <code>AxiomSoapMessageFactory</code>

Поскольку сообщения, посылаемые веб-службе оценки комбинации карт при игре в покер и принимаемые от нее, достаточно просты, я решил внедрить в свойство `messageFactory` компонента `WebServiceTemplate` объект `SaajSoapMessageFactory`. (Он также играет роль фабрики сообщений, используемой компонентом `MessageDispatcherServlet` по умолчанию.) Если впоследствии я решу перейти на более эффективную реализацию AXIOM, мне достаточно будет просто внедрить в свойство `messageFactory` другой объект.

Не только SOAP. На рис. 15.10 допущена небольшая неточность. Глядя на представленную схему, можно заключить, что Spring-WS может

взаимодействовать с веб-службами только по протоколу SOAP. В действительности же Spring-WS применяет SOAP только при использовании фабрики AxiomSoapMessageFactory или SaajSoapMessageFactory. Фабрика DomPoxMessageFactory поддерживает POX-сообщения, которые не упаковываются в пакеты SOAP. Если вы испытываете предубеждение к протоколу SOAP, тогда используйте фабрику DomPoxMessageFactory. Возможно, вам интересно будет узнать, что грядущая версия Spring-WS 1.1 будет также включать поддержку REST.

В свойство messageSender должна быть внедрена ссылка на реализацию интерфейса WebServiceMessageSender. И снова Spring-WS предоставляет пару соответствующих реализаций, которые перечислены в табл. 15.5.

Таблица 15.5. Отправители сообщений реализуют отправку сообщений веб-службе. Фреймворк Spring-WS содержит две реализации отправителей сообщений

Отправитель сообщений	Описание
CommonsHttpMessageSender	Отправляет сообщения с помощью библиотеки Jakarta Commons HTTP Client. Поддерживает предварительно настроенные HTTP-клиенты и такие дополнительные особенности, как аутентификация средствами HTTP и организация HTTP-соединений в пулы
HttpURLConnectionMessageSender	Отправляет сообщения с помощью базовой поддержки протокола HTTP в языке Java. Обладает ограниченной функциональностью

Выбор между CommonsHttpMessageSender и HttpURLConnectionMessageSender – это компромисс между функциональностью и зависимостью от сторонних JAR-файлов. Если приложению не требуются дополнительные возможности, поддерживаемые классом CommonsHttpMessageSender (такие как HTTP-аутентификация), возможностей класса HttpURLConnectionMessageSender будет вполне достаточно. Но если эти возможности необходимы, тогда придется использовать CommonsHttpMessageSender, при этом необходимо включить библиотеку Jakarta Commons HTTP в библиотеку классов (classpath) клиентского приложения.

Так как для работы с веб-службой оценки комбинации карт при игре в покер дополнительные возможности не потребуются, я выбрал HttpURLConnectionMessageSender, который в контексте Spring настраивается, как показано ниже:

```
<bean id="messageSender"
      class="org.springframework.ws.transport.http.
          ↳ HttpURLConnectionMessageSender">
    <property name="url"
              value="http://localhost:8080/Poker-WS/services"/>
</bean>
```

Свойство `url` определяет местоположение службы. Обратите внимание, что значение этого свойства совпадает с URL в WSDL-определении службы.

Если позднее потребуется реализовать аутентификацию пользователей веб-службы, достаточно будет просто изменить определение класса в компоненте `messageSender` на `CommonsHttpMessageSender`.

Отправка сообщения

После настройки компонента `WebServiceTemplate` его можно использовать для отправки и приема XML-сообщений. `WebServiceTemplate` предоставляет несколько методов для отправки и приема сообщений. Однако из них выделяется один, самый основной и простой в понимании:

```
public boolean sendAndReceive(Source requestPayload,
                               Result responseResult)
                               throws IOException
```

В качестве параметров метод `sendAndReceive()` принимает объекты `java.xml.transform.Source` и `java.xml.transform.Result`. Объект `Source` представляет содержимое сообщения, отправляемого веб-службе, а объект `Result` заполняется содержимым сообщения, возвращаемым веб-службой.

В листинге 15.5 представлен класс `TemplateBasedPokerClient`, реализующий интерфейс `PokerClient`, который использует метод `sendAndReceive()` класса `WebServiceTemplate` для взаимодействия со службой оценки комбинации карт при игре в покер.

Листинг 15.5. Клиент, использующий внедренный объект `WebServiceTemplate` для отправки и приема XML-сообщений

```
package com.springinaction.ws.client;
import java.io.IOException;
import org.jdom.Document;
import org.jdom.Element;
```

```
import org.jdom.Namespace;
import org.jdom.transform.JDOMResult;
import org.jdom.transform.JDOMSource;
import org.springframework.ws.client.core.WebServiceTemplate;
import com.springinaction.poker.Card;
import com.springinaction.poker.PokerHandType;

public class TemplateBasedPokerClient
    implements PokerClient {

    public PokerHandType evaluateHand(Card[] cards)
        throws IOException {

        // Конструирование XML-сообщения
        Element requestElement =
            new Element("EvaluateHandRequest");
        Namespace ns = Namespace.getNamespace(
            "http://www.springinaction.com/poker/schemas");
        requestElement.setNamespace(ns);
        Document doc = new Document(requestElement);

        for(int i=0; i<cards.length; i++) {
            Element cardElement = new Element("card");
            Element suitElement = new Element("suit");
            suitElement.setText(cards[i].getSuit().toString());
            Element faceElement = new Element("face");
            faceElement.setText(cards[i].getFace().toString());
            cardElement.addContent(suitElement);
            cardElement.addContent(faceElement);
            doc.getRootElement().addContent(cardElement);
        }

        // Отправка сообщения с использованием шаблона
        JDOMSource requestSource = new JDOMSource(doc);
        JDOMResult result = new JDOMResult();
        webServiceTemplate.sendAndReceive(requestSource, result);

        // Парсинг XML-ответа
        Document resultDocument = result.getDocument();
        Element responseElement = resultDocument.getRootElement();
        Element handNameElement =
            responseElement.getChild("handName", ns);
        return PokerHandType.valueOf(handNameElement.getText());
    }

    private WebServiceTemplate webServiceTemplate;
```

```
public void setWebServiceTemplate(          // Используется для
    WebServiceTemplate webServiceTemplate) { // внедрения шаблона
    this.webServiceTemplate = webServiceTemplate;
}
```

И Source, и Result – это интерфейсы, являющиеся стандартной частью Java XML API и доступные в Java SDK. Существует бесконечное множество реализаций этих интерфейсов, но, как можно заметить в листинге 15.5, я выбрал реализации, основанные на модели JDOM. Выбор был сделан достаточно случайно, но под влиянием моего знакомства с JDOM и знания особенностей использования этой модели для конструирования XML-сообщений.

Метод evaluateHand() класса TemplateBasedPokerClient начинается с создания сообщения <EvaluateHandRequest> из исходного массива элементов Card. После создания сообщения вызывается метод sendAndReceive() класса WebServiceTemplate. А затем выполняется парсинг результатов с преобразованием их в объект PokerHandType, который должен быть возвращен вызывающей программе.

Обратите внимание, что объект WebServiceTemplate внедряется через параметр метода записи. Отсюда следует, что компонент TemplateBasedPokerClient должен быть настроен следующим образом:

```
<bean id="templateBasedClient"
      class="com.springinaction.ws.client.TemplateBasedPokerClient">
    <property name="webServiceTemplate" ref="webServiceTemplate" />
</bean>
```

В свойство webServiceTemplate внедряется ссылка на компонент webServiceTemplate, который был сконфигурирован выше.

Просматривая листинг 15.5, можно заметить, что значительная часть метода evaluateHand() связана с созданием и парсингом XML-сообщений – собственно отправку сообщения выполняет единственная строка. Возможно, нет ничего плохого в том, что простые XML-сообщения будут создаваться и анализироваться вручную, но представьте себе, как вырастет объем кода, если потребуется конструировать достаточно сложные сообщения. Даже в случае с нашей службой, использующей далеко не сложные сообщения, объем кода, занимающегося обработкой XML, достаточно велик.

К счастью, можно вообще избавить себя от необходимости работать с XML. В разделе 15.4.4 было показано, как можно преобразо-

вывать объекты в формат XML и обратно с помощью маршалера. Теперь я покажу, как `WebServiceTemplate` может пользоваться услугами маршалеров, чтобы избавиться от необходимости вручную писать код обработки XML-сообщений.

Использование маршалеров на стороне клиента

Кроме простого метода `sendAndReceive()`, использованного в листинге 15.5, класс `WebServiceTemplate` предоставляет также метод `marshalSendAndReceive()`, реализующий отправку и прием XML-сообщений, преобразуемых в Java-объекты и обратно.

При использовании метода `marshalSendAndReceive()` достаточно просто передать ему объект запроса в виде параметра и принять объект ответа в виде возвращаемого значения. В случае со службой оценки комбинации карт при игре в покер этими объектами будут экземпляры `EvaluateHandRequest` и `EvaluateHandResponse` соответственно.

Листинг 15.6 демонстрирует определение класса `MarshallingPokerClient`, реализующего интерфейс `PokerClient` и использующего метод `marshalSendAndReceive()` для взаимодействий со службой.

Листинг 15.6. Класс MarshallingPokerClient, пользующийся услугами маршалера для преобразования объектов в формат XML и обратно

```
package com.springinaction.ws.client;
import java.io.IOException;
import org.springframework.ws.client.core.WebServiceTemplate;
import com.springinaction.poker.Card;
import com.springinaction.poker.PokerHandType;
import com.springinaction.poker.webservice.EvaluateHandRequest;
import com.springinaction.poker.webservice.EvaluateHandResponse;

public class MarshallingPokerClient
    implements PokerClient {

    public PokerHandType evaluateHand(Card[] cards)
        throws IOException {

        EvaluateHandRequest request = new EvaluateHandRequest(); // Создание
        request.setHand(cards); // запроса

        EvaluateHandResponse response = (EvaluateHandResponse) // Отправка
            webServiceTemplate.marshalSendAndReceive(request); // запроса

        return response.getPokerHand(); // Возвращает объект ответа
    }
}
```

```
}

private WebServiceTemplate webServiceTemplate;
public void setWebServiceTemplate(
    WebServiceTemplate webServiceTemplate) {
    this.webServiceTemplate = webServiceTemplate;
}
}
```

Ого! Метод `evaluateHand()` класса `MarshallingPokerClient` получилось намного проще и уже не содержит кода, реализующего обработку XML-сообщений. Вместо этого он конструирует объект `EvaluateHandRequest` и заполняет его массивом объектов `Card`. Затем вызывается метод `marshalSendAndReceive()`, принимающий объект `EvaluateHandRequest` и возвращающий объект `EvaluateHandResponse`, из которого затем извлекается в объект `PokerHandType` и возвращается вызывающей программе.

Но как компонент `WebServiceTemplate` узнает, каким образом выполнять маршалинг/демаршалинг объектов `EvaluateHandRequest` и `EvaluateHandResponse`? Неужели он настолько умный?

Ну... нет.., действительно нет. Он ничего не знает ни о маршалинге, ни о демаршалинге этих объектов. Однако, как показано на рис. 15.11, его можно связать с маршалером и демаршалером, которые знают, как выполнять маршалинг:

```
<bean id="webServiceTemplate"
    class="org.springframework.ws.client.core.WebServiceTemplate">
    <property name="messageFactory">
        <bean class="org.springframework.ws.soap.saaj.
            ↗ SaajSoapMessageFactory"/>
    </property>
    <property name="messageSender" ref="urlMessageSender"/>

    <property name="marshaller" ref="marshaller" />
    <property name="unmarshaller" ref="marshaller" />
</bean>
```

Здесь в оба свойства, `marshaller` и `unmarshaller`, я внедрил ссылку на компонент `marshaller`, который является тем же самым компонентом `CastorMarshaller`, который настраивался в разделе 15.4.4. Но на его месте мог бы быть любой другой маршалер из перечисленных в табл. 15.3.

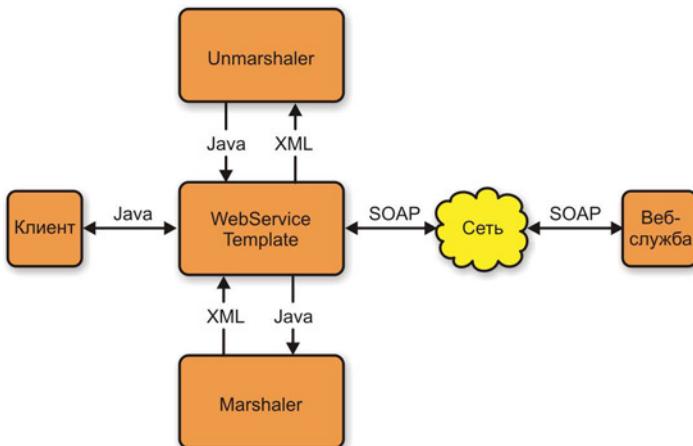


Рис. 15.11. При внедрении маршалера и демаршалера клиент получает возможность посыпать и принимать Java-объекты от компонента WebServiceTemplate. WebServiceTemplate будет использовать маршалер и демаршалер для преобразования Java-объектов в формат XML и обратно

Класс MarshallingPokerClient получился намного прозрачнее, чем TemplateBasedPokerClient. Но у нас есть возможность упростить клиента еще больше. Посмотрим далее, как с помощью класса WebServiceGatewaySupport из фреймворка Spring-WS избавиться от необходимости явно связывать WebServiceTemplate.

15.5.2. Использование поддержки шлюза веб-служб

Как рассказывалось в главе 8 (см. разделы 8.3.3, 8.4.3, 8.5.3 и 8.6.2), API доступа к данным в Spring включает ряд вспомогательных классов, предоставляющих шаблоны, которые не требуется настраивать. В Spring-WS имеется аналогичный класс поддержки WebServiceGatewaySupport, который автоматически предоставляет своим наследникам доступ к объекту WebServiceTemplate.

В листинге 15.7 демонстрируется окончательная реализация интерфейса PokerClient, класс PokerServiceGateway, наследующий класс WebServiceGatewaySupport.

Листинг 15.7. Класс WebServiceGatewaySupport предоставляет доступ к объекту WebServiceTemplate посредством метода getWebServiceTemplate()

```
package com.springinaction.ws.client;
import java.io.IOException;
import org.springframework.ws.client.core.support.WebServiceGatewaySupport;
import com.springinaction.poker.Card;
import com.springinaction.poker.PokerHandType;
import com.springinaction.poker.webservice.EvaluateHandRequest;
import com.springinaction.poker.webservice.EvaluateHandResponse;

public class PokerServiceGateway           // Наследует WebServiceGatewaySupport
    extends WebServiceGatewaySupport
    implements PokerClient {

    public PokerHandType evaluateHand(Card[] cards)
        throws IOException {
        EvaluateHandRequest request = new EvaluateHandRequest();

        request.setHand(cards);

        // Использовать предоставленный объект WebServiceTemplate
        EvaluateHandResponse response = (EvaluateHandResponse)
            getWebServiceTemplate().marshalSendAndReceive(request);

        return response.getPokerHand();
    }
}
```

Как видите, класс PokerServiceGateway мало отличается от класса MarshallingPokerClient. Основное отличие заключается в том, что PokerServiceGateway не требует внедрения компонента WebServiceTemplate. Вместо этого он получает экземпляр WebServiceTemplate вызовом метода getWebServiceTemplate(). За кулисами WebServiceGatewaySupport создает объект WebServiceTemplate без необходимости явно определять его в контексте Spring.

Даже при том, что больше не требуется явно определять компонент WebServiceTemplate в Spring, параметры создания экземпляра WebServiceTemplate все еще необходимо определять, но уже в виде свойств компонента WebServiceGatewaySupport. В случае использования

класса `PokerServiceGateway` это означает, что необходимо настроить свойства `messageFactory`, `messageSender`, `marshaller` и `unmarshaller`:

```
<bean id="pokerClientGateway"
      class="com.springinaction.ws.client.PokerServiceGateway">
    <property name="messageFactory">
      <bean class="org.springframework.ws.soap.saaj.SaajSoapMessageFactory"/>
    </property>
    <property name="messageSender" ref="messageSender"/>

    <property name="marshaller" ref="marshaller" />
    <property name="unmarshaller" ref="marshaller" />
</bean>
```

Обратите внимание, что свойствам присваиваются те же значения, что присваивались одноименным свойствам компонента `WebServiceTemplate`.

15.6. В заключение

Традиционно веб-службы рассматриваются как еще один способ организации удаленных взаимодействий. В действительности многие разработчики любовно называют их как «CORBA через XML».

Проблема применения веб-служб для организации удаленных взаимодействий обусловлена тем, что они влекут за собой образование тесной связи между службой и клиентами. Клиент оказывается накрепко привязанным к внутреннему API службы. А определение службы – это лишь побочный эффект такой связи. Изменения в службе могут вступить в противоречие с реализацией клиента, что потребует его изменения или поддержки нескольких версий службы.

В этой главе мы под разными углами зрения рассмотрели веб-службы, взаимодействие с которыми основано на обмене сообщениями. Этот подход к реализации веб-служб известен как модель «contract-first», поскольку в нем обслуживание строится на основе определения веб-службы. В отличие от простых удаленных объектов, веб-службы в модели «contract-first» реализуются как конечные точки, обрабатывающие сообщения, структура которых описывается определением службы. Как следствие служба и ее API могут изменяться, не нарушая определения.

Spring-WS – замечательный фреймворк для создания веб-служб в модели «contract-first». Опираясь на Spring MVC, конечные точки

Spring-WS обрабатывают XML-сообщения, полученные от клиента, и также генерируют в ответ XML-сообщения.

Если по складу характера вы похожи на меня, то наверняка весьма скептически восприняли всю проделанную выше работу по организации веб-службы с применением фреймворка Spring-WS. Не буду отрицать, что я в свое время отклонил модель «contract-first» создания веб-служб как требующую значительных усилий для SOAP-ифизации компонентов в сравнении с технологией JAX-WS, использующей модель «contract-last». Фактически, когда я впервые столкнулся с фреймворком Spring-WS, я отодвинул его в сторону как требующий слишком много работы и не обладающий явными преимуществами... какая глупость!

Но, немного поразмыслив, я понял, что преимущество слабой связанности определения службы с внутренним API легко перевешивает дополнительные усилия, которые требуется приложить при использовании Spring-WS. И эти усилия окупятся сторицей, так как в результате мы получаем возможность изменять и улучшать внутренний API приложения, не беспокоясь о нарушении договоренностей с клиентами, оформленными в виде определения.

В следующей главе мы увидим столкновение двух миров – поддержку использования существующих и разработки новых компонентов Enterprise JavaBeans.



Глава 16. Spring и Enterprise JavaBeans

В этой главе рассматриваются следующие темы:

- ❑ внедрение компонентов EJB в контекст Spring;
- ❑ конструирование компонентов EJB с поддержкой Spring;
- ❑ использование аннотаций EJB 3 с компонентами Spring.

Несколько лет тому назад на телевидении крутили ролик с рекламой арахисового масла компании Reese. В этом ролике один персонаж наслаждался чашкой горячего шоколада, а другой – чашкой арахисового масла. В какой-то момент персонажи сталкивались, и их напитки смешивались. Один из них воскликнул: «Вы плеснули шоколад в мое арахисовое масло!», а другой вторил ему: «Ваше арахисовое масло попало в мой шоколад!» После этого каждый пробовал получившуюся смесь, и оба приходили к выводу, что «два отличных вкуса при смешивании дают еще более приятный вкус».

Возможно, вас удивит наличие в этой книге раздела, описывающего, как использовать компоненты EJB совместно с фреймворком Spring. Большая часть книги была посвящена разработке приложений корпоративного уровня вообще без применения компонентов EJB. При сравнении Spring и EJB многие разработчики более склонны проводить аналогии с нефтью и водой, нежели с шоколадом и арахисовым маслом.

Несмотря на то что Spring предоставляет массу функциональных возможностей, которые способны поднять объекты POJO до уровня компонентов EJB, у вас не всегда может быть роскошь работать над проектом, полностью свободным от EJB. Кроме того, вам может потребоваться организовать взаимодействия с другими системами, экспортирующими свою функциональность посредством сеансовых компонентов EJB. Более того, вы можете оказаться в условиях, где одним из требований, по техническим или политическим причинам, является обязательное использование компонентов EJB.

Независимо от того, является ли ваше приложение клиентом, потребляющим услуги, поставляемые компонентами EJB, или вы сами вынуждены писать компоненты EJB, у вас нет причин полностью отказываться от преимуществ Spring. Фреймворк Spring предоставляет три способа смешивания Spring и EJB, чтобы получить выгоду от обеих архитектур.

- ❑ Если ваше приложение пользуется службами, основанными на сеансовых компонентах EJB, фреймворк Spring позволяет объявлять компоненты EJB как компоненты в контексте приложения Spring. Это дает возможность внедрять ссылки на сеансовые компоненты EJB (и EJB 2.x, и EJB 3) в свойства ваших компонентов, как если бы компоненты EJB были обычными POJO.
- ❑ Разрабатывая сеансовый компонент EJB 2.x, в нем можно предусмотреть поддержку фреймворка Spring. Благодаря этому ваш компонент EJB получит доступ к контексту приложения Spring и сможет использовать другие компоненты, настроенные в Spring.
- ❑ Фреймворк Pitchfork, обладающий поддержкой Spring, позволяет использовать аннотации EJB 3 для применения простейших приемов аспектно-ориентированного программирования и внедрения зависимостей в компоненты, сконфигурированные в контексте Spring.

В этой главе рассказывается об объединении компонентов EJB и Spring. Если у вас есть необходимость писать компоненты EJB с поддержкой Spring или внедрять EJB в свои Spring-приложения, эта глава для вас. Мы исследуем все доступные способы смешивания Spring и EJB, начав с внедрения компонентов EJB в контекст приложения Spring.

16.1. Внедрение компонентов EJB в Spring

Если прежде вам приходилось писать клиентские приложения для 2.x EJB, то вы наверняка знаете, как получить доступ к ссылке на компонент EJB. Сначала необходимо получить ссылку на домашний интерфейс компонента EJB из JNDI, как показано ниже:

```
private TrafficServiceHome trafficServiceHome;  
private TrafficServiceHome getTrafficServiceHome ()  
    throws javax.naming.NamingException {
```

```
if(trafficServiceHome != null)
    return trafficServiceHome;

javax.naming.InitialContext ctx =
    new javax.naming.InitialContext();

try {
    Object objHome = ctx.lookup("trafficService");

    TrafficServiceHome home =
        (TrafficServiceHome) javax.rmi.PortableRemoteObject.narrow(
            objHome, TrafficServiceHome.class);

    trafficServiceHome = home;
    return home;
} finally {
    ctx.close();
}
}
```

А затем получить ссылку на прикладной интерфейс компонента EJB (удаленный или локальный), чтобы иметь возможность вызывать его методы. Например, следующий фрагмент демонстрирует, как вызвать метод `getTrafficConditions()` службы управления движением EJB:

```
try {
    TrafficServiceHome home = getTrafficServiceHome();
    TrafficService trafficService = home.create();

    TrafficConditions conditions =
        trafficService.getTrafficConditions(city, state);
} catch (java.rmi.RemoteException e) {
    throw new TrafficException();
} catch (CreateException e) {
    throw new TrafficException();
}
}
```

М-да! Такой объем кода, и только чтобы выяснить условия организации движения. Самое неприятное, что из всего этого лишь несколько строк имеют прямое отношение к получению информации. Большая же часть – это шаблонный код, необходимый, только чтобы получить ссылку на компонент EJB. Такой объем работы, и только

чтобы совершить единственный вызов метода `getTrafficConditions()` компонента EJB.

Спецификация EJB 3 несколько улучшила положение дел. Вместо получения ссылки на домашний интерфейс компонента EJB, из JNDI можно сразу получить сеансовый компонент EJB 3. Но и в этом случае приходится писать массу шаблонного кода, реализующего операции с JNDI.

Но постойте! На протяжении всей книги мы не раз сталкивались с разными способами внедрения в прикладные компоненты ссылок на службы, в которых они нуждаются. При этом компоненты не занимались поиском других компонентов – компоненты *внедрялись* в иные компоненты посредством конфигурации. А этот пример практически целиком посвящен поиску компонента EJB и его домашнего интерфейса в JNDI, что совершенно не укладывается в наши представления. Если мы продолжим работать с компонентами EJB традиционным способом, мы в результате получим массу уродливого кода, реализующего поиск и образующего тесную связь с EJB. Нет ли более простого пути?

16.1.1. Проксирование сеансовых компонентов (EJB 2.x)

Как вы уже, вероятно, догадались из вступления, более простой путь действительно существует. В главе 11 было показано, как настроить прокси-объекты для организации доступа к различным удаленным службам, включая службы, основанные на RMI, Hessian, Burlap и Spring HTTP Invoker. Практически те же самые возможности предлагаются фреймворком Spring и для доступа к компонентам EJB.

В Spring имеются два фабричных компонента, производящих прокси-объекты, пригодные для работы с сеансовыми компонентами EJB:

- ❑ `LocalStatelessSessionProxyFactoryBean` – используется для доступа к локальным компонентам EJB (находящимся в том же контейнере, что и клиент);
- ❑ `SimpleRemoteStatelessSessionProxyFactoryBean` – используется для доступа к удаленным компонентам EJB (находящимся в другом контейнере, отличном от того, где располагается клиент).

Как показано на рис. 16.1, эти фабричные компоненты создают прокси-объекты, автоматически отыскивающие домашние интер-

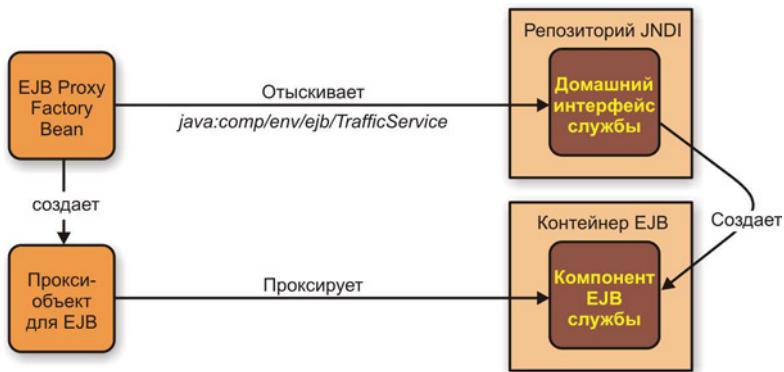


Рис. 16.1. Фабричный компонент, производящий прокси-объекты для доступа к EJB, автоматически отыскивает домашний интерфейс компонента и производит прокси-объект, делегирующий вызовы методов фактическому компоненту EJB

фейсы компонентов EJB и вызывающие методы их прикладных интерфейсов. Они позволяют определять ссылки на компоненты EJB в контексте приложения Spring, которые затем могут внедряться, как если бы они были ссылками на обычные компоненты в контексте Spring.

Для простоты демонстрации предположим, что компонент EJB службы управления дорожным движением является локальным сеансовым компонентом EJB. Чтобы внедрить его в Spring, можно воспользоваться компонентом `LocalStatelessSessionProxyFactoryBean`, как показано ниже:

```

<bean id="trafficService"
      class="org.springframework.ejb.access.
           LocalStatelessSessionProxyFactoryBean"
      lazy-init="true">

    <property name="jndiName" value="ejb/TrafficService" />
    <property name="businessInterface"
              value="com.rodrantz.ejb.TrafficServiceEjb" />
</bean>
  
```

Свойство `jndiName` компонента `trafficService` в этом примере используется для идентификации имени домашнего интерфейса EJB

в репозитории JNDI. А свойство `businessInterface` идентифицирует прикладной интерфейс компонента EJB. Этот интерфейс будет поддерживаться прокси-объектом.

Обратите особое внимание на атрибут `lazy-init` в элементе `<bean>`. Контекст приложения Spring обычно заранее создает экземпляры одиночных компонентов в момент загрузки конфигурационного файла Spring. В большинстве случаев это правильное решение, но при использовании прокси-объектов для доступа к EJB это может вызывать проблемы, из-за того что контекст приложения Spring может загрузить и создать прокси-объект еще до того, как будет получен домашний интерфейс компонента EJB. Определив атрибут `lazy-init` со значением `true`, мы сообщаем фреймворку Spring, что он не должен пытаться получить домашний интерфейс до первого обращения к компоненту `trafficService`.

Теперь отвлечемся и посмотрим, как можно было бы настроить компонент `trafficService`, если бы служба управления движением была реализована в виде удаленного сеансового компонента. Взглядите на следующий фрагмент разметки XML:

```
<bean id="trafficService"
      class="org.springframework.ejb.access.
           SimpleRemoteStatelessSessionProxyFactoryBean"
      lazy-init="true">

    <property name="jndiName" value="trafficService" />
    <property name="businessInterface"
              value="com.rodrantz.ejb.TrafficServiceEjb" />
</bean>
```

Видите разницу? Единственное, что изменилось, – имя класса фабричного компонента. Все остальное осталось, как и прежде. Фреймворк Spring делает выбор между локальными и удаленными компонентами EJB практически прозрачным.

Возможно, кого-то волнует вопрос обработки исключения `java.rmi.RemoteException`. Как выбор между локальными и удаленными компонентами EJB может быть полностью прозрачным, если вызов метода удаленного компонента EJB может возбудить исключение `RemoteException`? Разве не нужно где-то перехватить это исключение?

В этом заключается еще одно преимущество использования поддержки EJB в Spring для доступа к компонентам EJB. Как и при работе со службами на основе механизма RMI, любые исключения



RemoteException, которые могут возбуждаться компонентами EJB, будут перехвачены и преобразованы в исключение org.springframework.remoting.RemoteAccessException. А поскольку исключение RemoteAccessException является неконтролируемым, его можно не перехватывать.

Объявление прокси-объектов для EJB с помощью XML-элементов из пространства имен jee

Фабричные компоненты из фреймворка Spring, используемые для организации доступа к компонентам EJB, значительно упрощают работу с EJB и позволяют внедрять их в компоненты Spring, как если бы они были обычными компонентами Spring. Но можно ли еще более упростить жизнь?

Да, конечно! В версии Spring 2, в новом пространстве имен jee, появились конфигурационные элементы, еще более упрощающие настройку компонентов EJB. Пространство имен jee включает два конфигурационных элемента, специализирующихся на настройке EJB:

- ❑ <jee:local-slsb> – настраивает прокси-объект для локального сеансового компонента EJB в контексте приложения Spring;
- ❑ <jee:remote-slsb> – настраивает прокси-объект для удаленного сеансового компонента EJB в контексте приложения Spring.

Чтобы воспользоваться этими двумя элементами, необходимо добавить объявление пространства имен jee в конфигурационный файл Spring, включив следующий элемент <beans>:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
        http://www.springframework.org/schema/jee
        http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">
```

После объявления пространства имен jee можно приступать к использованию его элементов для настройки ссылок на компоненты EJB в Spring. Например, чтобы настроить ссылку на локальный сеансовый компонент EJB, можно воспользоваться элементом <jee:local-slsb>, как показано ниже:

```
<jee:local-slsb id="trafficService"
    jndi-name="trafficService"
    business-interface="com.rodrantz.ejb.TrafficServiceEjb"/>
```

За кулисами элемент `<jee:local-slsb>` автоматически настроит компонент `LocalStatelessSessionProxyFactoryBean` в контексте Spring. Конечный результат будет тем же, а объем разметки XML – меньше.

Аналогично, с помощью элемента `<jee:remote-slsb>`, можно внедрить удаленный сеансовый компонент EJB:

```
<jee:remote-slsb id="trafficService"
    jndi-name="trafficService"
    business-interface="com.rodrantz.ejb.TrafficServiceEjb"/>
```

Как элемент `<jee:local-slsb>` является более краткой формой настройки компонента `LocalStatelessSessionProxyFactoryBean`, так и `<jee:remote-slsb>` является более краткой формой настройки компонента `SimpleRemoteStatelessSessionProxyFactoryBean`.

Объявление сеансовых компонентов EJB 3 в Spring

Как упоминалось выше, спецификация EJB 3 упрощает процесс получения ссылки на компоненты, избавляя от необходимости приобретать их домашние интерфейсы. Вместо получения ссылки на сеансовый компонент посредством его домашнего интерфейса, извлекаемого из JNDI, ссылки на сеансовые компоненты EJB 3 извлекаются из JNDI непосредственно.

Но, как вы уже видели, реализация поиска компонентов в JNDI выглядит достаточно сложно и по большей части является шаблонной. Кроме того, организация извлечения EJB из репозитория идет вразрез с принципом внедрения зависимостей, которому следует фреймворк Spring. В Spring сеансовые компоненты должны внедряться, а не извлекаться.

К счастью, фреймворк Spring предоставляет возможность внедрения объектов, хранящихся в JNDI, как любых других компонентов, имеющихся в контексте приложения. Вся хитрость заключается в использовании компонента `JndiObjectFactoryBean` из фреймворка Spring. Следующий фрагмент конфигурационного XML-файла демонстрирует объявление сеансового компонента EJB 3 службы управления движением:

```
<bean id="trafficService"
    class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="ejb/TrafficService" />
    <property name="resourceRef" value="true" />
</bean>
```

Будучи фабричным компонентом, JndiObjectFactoryBean производит прокси-объект, обеспечивающий доступ к фактическому сеансовому компоненту (как показано на рис. 16.2), поиск которого выполняется в репозитории JNDI по имени, указанному в свойстве jndiName. Свойство resourceRef указывает, что компонент EJB должен извлекаться как Java-ресурс, вследствие чего перед выполнением поиска к значению свойства jndiName будет добавлен префикс java:comp/env/.



Рис. 16.2. Сеансовые компоненты EJB 3 можно также настраивать с использованием прокси-объектов для доступа к JNDI

При использовании пространства имён jee компонент EJB можно также объявить с помощью элемента <jee:jndi-lookup>:

```
<jee:jndi-lookup id="trafficService"
    jndi-name="ejb/TrafficService"
    resource-ref="true" />
```

Этот фрагмент эквивалентен объявлению <bean> выше. За кулисами элемент <jee:jndi-lookup> создает компонент JndiObjectFactoryBean. Подробнее об элементе <jee:jndi-lookup> будет рассказываться в главе 17. А пока достаточно знать, что JndiObjectFactoryBean создает прокси-объект для доступа к сеансовым компонентам, хранящимся в JNDI.

Теперь, после объявления прокси-объекта для доступа к сеансовому компоненту, можно включить его в работу. Посмотрим, как внедрить EJB в POJO, сконфигурированный в Spring.

16.1.2. Внедрение компонентов EJB в компоненты Spring

Как оказывается, внедрение EJB в POJO, сконфигурированный в Spring, ничем не отличается от внедрения любых других компонентов. Например, внедрить сеансовый компонент службы управления движением в компонент rantService можно следующим способом:

```
<bean id="rantService"
      class="com.roadrantz.service.RantServiceImpl">
    ...
    <property name="trafficService" ref="trafficService" />
    ...
</bean>
```

Заметили? Здесь нет ничего, что говорило бы об использовании компонента EJB. Здесь в свойство trafficService просто внедряется компонент trafficService (который по стечению обстоятельств является прокси-объектом для доступа к EJB). Здесь вообще отсутствуют какие-либо признаки, свидетельствующие, что речь идет о компоненте EJB. Как показано на рис. 16.3, проксируемые компоненты EJB могут внедряться в другие компоненты, подобно любым другим POJO, сконфигурированным в Spring.

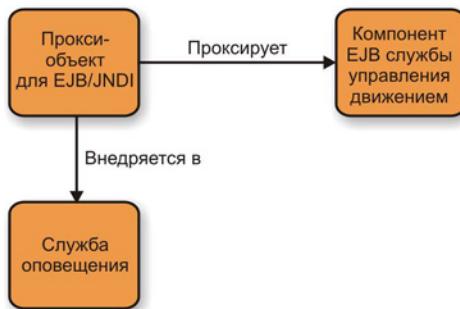


Рис. 16.3. Прокси-объекты для доступа к EJB и JNDI могут внедряться в компоненты Spring, как если бы они были обычными компонентами Spring

Вся прелесть использования фабричных компонентов для организации доступа к компоненту EJB службы управления движением состоит в том, что нет необходимости писать собственный код, выполняющий поиск этой службы, или код, делегирующий вызовы прикладных методов. Фактически нет даже необходимости писать код, взаимодействующий с JNDI или использующий домашний интерфейс компонента EJB.

Кроме того, благодаря тому что все это скрыто за прикладным интерфейсом TrafficService, компонент trafficService даже понятия не имеет, что взаимодействует с компонентом EJB. С его точки зрения



он использует другой POJO. Это важно, потому что подразумевает возможность замены EJB-реализации интерфейса TrafficService любой другой реализацией (возможно, даже фиктивной, используемой для тестирования класса RantServiceImpl).

Теперь, когда вы познакомились с особенностями внедрения компонентов EJB в приложения на основе фреймворка Spring, посмотрим, как Spring поддерживает разработку новых компонентов EJB.

16.2. Разработка компонентов с поддержкой Spring (EJB 2.x)

Фреймворк Spring предоставляет массу возможностей для разработки корпоративных приложений без использования компонентов EJB, тем не менее, у вас все еще может возникать необходимость создавать собственные компоненты EJB.

В главе 11 демонстрировалось, как компоненты-экспортеры, входящие в состав Spring позволяют превратить любой POJO в удаленную службу. Я очень не хочу разочаровывать вас, но, к сожалению, фреймворк Spring не содержит класса EjbServiceExporter, который экспорттировал бы объекты POJO как компоненты EJB. (Но я соглашусь, что иметь такой класс было бы здорово.)

Однако, как бы то ни было, фреймворк Spring помогает немного облегчить разработку компонентов EJB. В состав Spring входят четыре абстрактных класса, которые привносят обычные компоненты EJB в мир Spring:

- ❑ AbstractMessageDrivenBean – может пригодиться для разработки компонентов, управляемых сообщениями, которые принимают сообщения из источников, отличных от JMS (согласно спецификации EJB 2.1);
- ❑ AbstractJmsMessageDrivenBean – может пригодиться для разработки компонентов, управляемых сообщениями, которые принимают сообщения из JMS-источников;
- ❑ AbstractStatefulSessionBean – может пригодиться для разработки сеансовых компонентов EJB с поддержкой информации о состоянии;
- ❑ AbstractStatelessSessionBean – может пригодиться для разработки сеансовых компонентов EJB, не поддерживающих информацию о состоянии.

Эти абстрактные классы упрощают разработку компонентов EJB двумя способами.

- ❑ Предоставляют пустые реализации методов управления жизненным циклом EJB (например, ejbActivate(), ejbPassivate(), ejbRemove()). Спецификация EJB требует наличия этих методов, но обычно они имеют пустые реализации.
- ❑ Предоставляют доступ к фабрике компонентов Spring. Это позволяет реализовать компоненты EJB, делегирующие реализацию прикладной логики объектам POJO, сконфигурированным в контексте Spring. Фактически компонент EJB может являться лишь фасадом для объектов POJO в контексте Spring.

Для примера предположим, что необходимо экспорттировать функциональность компонента rantService в виде сеансового компонента EJB без поддержки информации о состоянии. В листинге 16.1 показано, как может выглядеть реализация такого компонента EJB.

Листинг 16.1. Сеансовый компонент EJB без поддержки информации о состоянии, делегирующий реализацию прикладной логики объекту POJO в Spring

```
package com.roadrantz.ejb;
import java.util.Date;
import java.util.List;
import javax.ejb.CreateException;
import org.springframework.ejb.support.AbstractStatelessSessionBean;
import com.roadrantz.domain.Motorist;
import com.roadrantz.domain.Rant;
import com.roadrantz.domain.Vehicle;
import com.roadrantz.service.MotoristAlreadyExistsException;
import com.roadrantz.service.RantService;

public class RantServiceEjb
    extends AbstractStatelessSessionBean
    implements RantService {
    public RantServiceEjb() {}

    private RantService rantService;
    protected void onEjbCreate() throws CreateException {
        rantService = (RantService)           // Поиск службы оповещения
                      getBeanFactory().getBean("rantService");
    }

    public void addMotorist(Motorist motorist)
        throws MotoristAlreadyExistsException {
        rantService.addMotorist(motorist);
```

```
}

// Следующие методы делегируют выполнение прикладной логики объекту POJO
public void addRant(Rant rant) {
    rantService.addRant(rant);
}

public List<Rant> getRantsForDay(Date date) {
    return rantService.getRantsForDay(date);
}

public List<Rant> getRantsForVehicle(Vehicle vehicle) {
    return rantService.getRantsForVehicle(vehicle);
}

public List<Rant> getRecentRants() {
    return rantService.getRecentRants();
}

public void sendDailyRantEmails() {
    rantService.sendDailyRantEmails();
}

public void sendEmailForVehicle(Vehicle vehicle) {
    rantService.sendEmailForVehicle(vehicle);
}
}
```

После создания экземпляра RantServiceEjb его метод onEjbCreate() извлечет компонент rantService из фабрики компонентов Spring. Затем, когда будет вызван какой-либо из методов, он вызовет соответствующий метод компонента rantService, как показано на рис. 16.4.

Листинг 16.1 не дает ответа на один большой вопрос – откуда берется ссылка на фабрику компонентов. В типичной манере для JEE абстрактные классы компонентов EJB извлекают фабрику компонентов из JNDI. По умолчанию они пытаются отыскать фабрику компонентов по имени `java:comp/env/ejb/BeanFactoryPath`. Это означает необходимость настройки в JNDI фабрики компонентов с этим именем.

Если у вас фабрика компонентов хранится в JNDI с другим именем, определите свойство `beanFactoryLocatorKey` перед загрузкой фабрики компонентов (либо в конструкторе, либо в методе `setSessionContext()`). Например:

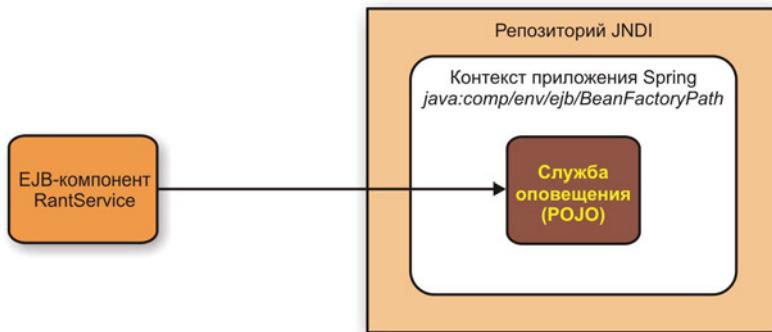


Рис. 16.4. Поддержка создания компонентов EJB в Spring имеет форму абстрактных классов, позволяющих разрабатывать компоненты EJB, имеющие доступ к контексту приложения Spring (хранящемуся в JNDI)

```
public void setSessionContext(SessionContext sessionContext) {  
    super.setSessionContext(sessionContext);  
  
    setBeanFactoryLocatorKey("java:comp/env/ejb/SpringContext");  
}
```

При такой реализации метода `setSessionContext()` контекст Spring должен храниться в JNDI с именем `java:comp/env/ejb/SpringContext`.

Поддержка разработки компонентов EJB в Spring основана на спецификациях EJB 2.x. Однако спецификация EJB 3 содержит существенные изменения. Чтобы упростить разработку компонентов EJB, спецификация EJB 3 заимствовала из Spring некоторые идеи, такие как внедрение зависимостей и аспектно-ориентированное программирование. Поэтому далее познакомимся со спецификацией EJB 3 поближе и посмотрим, как она совмещается с фреймворком Spring.

16.3. Spring и EJB3

Компоненты EJB пользуются большой популярностью в среде Java-разработчиков с самого момента их появления, тем не менее они имеют свои *сложности*:

- ❑ получение доступа к EJB связано с необходимостью реализации взаимодействия с JNDI, извлечения домашнего интерфей-



са компонента и последующего его использования для создания прикладного интерфейса;

- ❑ согласно спецификации EJB 2.x, компоненты EJB должны реализовать интерфейсы управления жизненным циклом, определяющие методы обратного вызова, которые в большинстве приложений не используются и имеют пустые реализации;
- ❑ сигнатуры методов удаленных компонентов EJB должны включать определение `throw java.rmi.RemoteException`, даже если реализация метода не предусматривает возбуждения исключения.

Эти и другие проблемы способствовали потере интереса к компонентам EJB у многих разработчиков и вынудили их искать более простые альтернативы, такие как фреймворк Spring. Реагируя на отрицательные тенденции, организация Java Community Process пересмотрела спецификацию EJB и внесла существенные изменения, представив сообществу новую спецификацию: EJB 3.

Спецификация EJB 3 решает проблемы своей тяжеловесной предшественницы, предусмотрев поддержку внедрения зависимостей для компонентов EJB и ресурсов взамен сложных взаимодействий с JNDI. При этом EJB 3 приветствует использование аннотаций Java 5 для объявления зависимостей, которые должны внедряться в свойства компонентов.

Кроме того, спецификация EJB 3 не требует, чтобы компоненты EJB реализовали какой-то специализированный интерфейс или методы управления жизненным циклом. А для удаленных методов больше не требуется объявлять, что они возбуждают исключение `RemoteException`. Проще говоря, спецификация EJB 3 приняла модель программирования на основе POJO.

В Spring отсутствует непосредственная поддержка спецификации EJB 3. Однако для Spring имеется расширение, позволяющее использовать аннотации EJB 3 для внедрения зависимостей и использования приемов аспектно-ориентированного программирования.

16.3.1. Pitchfork

Фреймворк Pitchfork – это расширение для Spring, обеспечивающее поддержку аннотаций EJB 3. Это совместная разработка Interface 21 (коллектив разработчиков Spring) и BEA, и используется в WebLogic Server 10 компании BEA для поддержки EJB 3. Но, чтобы использовать фреймворк Pitchfork, совсем необязатель-

но пользоваться WebLogic. Pitchfork – открытое программное обеспечение, распространяемое на условиях лицензии Apache Software License 2.0, может использоваться в любых приложениях, основанных на фреймворке Spring 2.0. Загрузить Pitchfork можно на сайте компании Interface 21 <http://www.springframework.com/pitchfork>.

Фреймворк Pitchfork не является полноценной реализацией спецификации EJB 3. Однако он поддерживает внедрение зависимостей и аспектно-ориентированное программирование посредством аннотаций EJB 3, включая перечисленные в табл. 16.1.

В этом разделе будет показано, как с помощью Pitchfork можно использовать некоторые из этих аннотаций в контексте Spring. При этом будет предполагаться, что вы уже знакомы с положениями спецификации EJB 3 и этими аннотациями. Более подробное обсуждение EJB 3 можно найти в книге «EJB 3 in Action» (Manning, 2006).

Таблица 16.1. Аннотации EJB 3, поддерживаемые фреймворком Pitchfork

Аннотация	Описание
@ApplicationException	Объявляет исключение прикладным исключениям, которое по умолчанию не будет откатывать транзакцию
@AroundInvoke	Объявляет метод методом перехватчика
@EJB	Объявляет зависимость от EJB
@ExcludeClassInterceptors	Объявляет, что метод не должен перехватываться классом-перехватчиком
@ExcludeDefaultInterceptors	Объявляет, что метод не должен перехватываться классом-перехватчиком, используемым по умолчанию
@Interceptors	Определяет один или более классов-перехватчиков для связи с классом компонента или методом
@PostConstruct	Определяет метод, который будет вызван для завершения инициализации после создания компонента и внедрения всех зависимостей
@PreDestroy	Определяет метод, который будет вызван перед удалением компонента из контейнера
@Resource	Объявляет зависимость от внешнего ресурса
@Stateless	Объявляет, что компонент будет сеансовым компонентом без поддержки информации о состоянии
@TransactionAttribute	Определяет, что метод должен вызываться в контексте транзакции

16.3.2. Введение в Pitchfork

Для внедрения зависимостей в компоненты, отмеченные аннотациями EJB 3, Pitchfork использует постпроцессор фабрики компонентов. На выбор предлагаются два постпроцессора:

- org.springframework.jee.config.JeeBeanFactoryPostProcessor;
- org.springframework.jee.ejb.config.JeeEjbBeanFactoryPostProcessor.

Эти два постпроцессора практически идентичны. Оба поддерживают все аннотации, перечисленные в табл. 16.1, кроме аннотации @EJB, которая поддерживается только постпроцессором JeeEjbBeanFactoryPostProcessor. Если вам необходима аннотация @EJB, выбирать следует постпроцессор JeeEjbBeanFactoryPostProcessor. В остальных случаях можно использовать любой из постпроцессоров.

Чтобы настроить один из этих постпроцессоров в Spring, достаточно просто добавить его как компонент в контекст приложения Spring. Например, чтобы задействовать JeeBeanFactoryPostProcessor, добавьте следующее объявление:

```
<bean class="org.springframework.jee.config.JeeBeanFactoryPostProcessor" />
```

После настройки JeeBeanFactoryPostProcessor можно начинать использовать аннотации EJB 3. Далее я покажу, как применять их к компонентам Spring.

16.3.3. Внедрение ресурсов с помощью аннотации

Для иллюстрации использования аннотаций EJB 3 с помощью Pitchfork вернемся к примеру класса рыцаря из главы 1. Представьте, что нам необходимо переписать класс BraveKnight из главы 1 и задействовать аннотацию @Resource для внедрения зависимости. Это можно сделать, как показано в листинге 16.2.

Листинг 16.2. Внедрение сценария в компонент BraveKnight с помощью аннотации

```
package com.springinaction.knight;
import javax.annotation.Resource;

public class BraveKnight implements Knight {
    @Resource(name = "quest")          // Внедрение сценария подвига
    private Quest quest;
```

```
public String name;

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

public String getName() {
    return name;
}

public void embarkOnQuest() {
    quest.embark(); // Использование внедренного сценария
}
}
```

Ниже демонстрируется, как выполняется настройка этого компонента `BraveKnight`:

```
<bean id="knight" class="com.springinaction.knight.BraveKnight">
    <constructor-arg value="Bedivere" />
</bean>
```

В главе 1 внедрение сценария выполнялось в XML-конфигурации. А здесь эта обязанность возложена на аннотацию `@Resource`. Аннотация `@Resource` попытается отыскать объект с именем `quest` и при обнаружении внедрит его в свойство `quest`. Обратите внимание, что здесь отпала надобность в методе `setQuest()` – аннотация `@Resource` способна внедрять зависимости непосредственно в частные свойства!

Но откуда будет взят объект `quest`? Сначала Pitchfork проверит наличие объекта `quest` в репозитории JNDI и, если такой объект имеется, внедрит в свойство `quest` объект из репозитория JNDI. Если поиск в JNDI не увенчается успехом, будет предпринята попытка отыскать компонент с именем `quest` в контексте приложения Spring.

Таким образом, вам необходимо будет либо поместить реализацию интерфейса `Quest` в JNDI, либо объявить компонент Spring с именем `quest`:

```
<bean id="quest"
      class="com.springinaction.knight.SlayDragonQuest" />
```

Мы познакомились с возможностью Pitchfork внедрять зависимости посредством аннотации EJB 3. А теперь посмотрим, как Pitchfork

поддерживает аспектно-ориентированное программирование EJB 3 на примере перехватчиков.

16.3.4. Объявление перехватчиков с помощью аннотаций

Помимо внедрения зависимостей, Pitchfork поддерживает также аннотации EJB 3 объявления перехватчиков. Перехватчики EJB 3 – это простейшая форма советов, выполняемых и до, и после вызова целевого метода, которые могут применяться с помощью аннотаций.

Например, класс Minstrel из главы 1 можно было бы переписать с применением аннотации @AroundInvoke, как показано в листинге 16.3.

Листинг 16.3. Класс менестреля, преобразованный в перехватчик с помощью аннотации EJB 3

```
package com.springinaction.knight;
// инструкции импортирования опущены

public class Minstrel {
    @AroundInvoke // Объявление перехватывающего метода
    public Object singAboutQuest(InvocationContext ctx)
        throws Exception {
        Knight knight = (Knight) ctx.getTarget();

        Logger song =
            Logger.getLogger(knight.getClass());

        Method method = ctx.getMethod();

        song.debug("Brave " + knight.getName() +
                  " did " + method.getName());

        Object rtn = ctx.proceed();      // Вызов целевого метода

        return rtn;
    }
}
```

Аннотация @AroundInvoke объявляет, что метод будет вызываться при перехвате вызова целевого метода. В данном случае метод singAboutQuest() будет вызываться перед вызовом целевого метода, чтобы менестрель смог воспеть подвиг рыцаря.

Сама аннотация `@AroundInvoke` лишь определяет метод-перехватчик. Поэтому нам необходимо как-то применить его к классу `BraveKnight`. Для этого можно воспользоваться аннотацией `@Interceptors`:

```
@Interceptors({Minstrel.class})
public class BraveKnight implements Knight {
    ...
}
```

Аннотация `@Interceptors` принимает массив из одного или более классов-перехватчиков (то есть классов, имеющих методы, отмеченные аннотацией `@AroundInvoke`). Когда класс отмечается аннотацией `@Interceptors`, вызовы всех его методов будут перехватываться классами-перехватчиками, указанными в списке. Поскольку класс `BraveKnight` имеет единственный метод `embarkOnQuest()`, перехватываться будут только вызовы этого метода. Однако, если потребуется уточнить, вызовы каких методов могут перехватываться, можно переместить аннотацию `@Interceptors` на уровень методов:

```
@Interceptors({Minstrel.class})
public void embarkOnQuest() {
    quest.embark();
}
```

Когда аннотация применяется к конкретному методу, перехватываться будут только вызовы аннотированных методов.

Другой способ ограничить перечень перехватываемых вызовов методов заключается в использовании аннотации `@ExcludeClassInterceptors`. При применении к конкретному методу аннотация `@ExcludeClassInterceptors` будет препятствовать указанным в ней классам-перехватчикам перехватывать вызовы этого метода. Например, чтобы предотвратить перехват вызовов метода `embarkOnQuest()`, его можно отметить аннотацией, как показано ниже:

```
@ExcludeClassInterceptors
public void embarkOnQuest() {
    quest.embark();
}
```

Фреймворк Pitchfork дает разработчикам приложений на основе Spring свободу выбора. Вы можете использовать обычный механизм



внедрения зависимостей и поддержку AOP в Spring или применять аннотации и AOP EJB 3. Однако практически в любых обстоятельствах средства фреймворка Spring выглядят предпочтительнее. Аспекты Spring AOP, например, обеспечивают большую гибкость, чем перехватчики EJB 3. Тем не менее фреймворк Pitchfork увеличивает свободу выбора.

16.4. В заключение

Модель программирования на основе POJO, поддерживаемая фреймворком Spring, представляет собой весьма привлекательную альтернативу компонентам Enterprise JavaBeans, тем не менее могут существовать факторы (технические, политические или исторические), вынуждающие отдать предпочтение компонентам EJB. В таких случаях совсем необязательно отказываться от использования Spring, так как Spring поддерживает разработку новых компонентов EJB и использование имеющихся.

В этой главе мы увидели, как обеспечить совместную работу Spring и EJB, начав с того, как превратить компоненты Spring в клиентов, пользующихся услугами компонентов EJB. С помощью прокси-объектов мы смогли внедрить ссылки на компоненты EJB в контекст приложения Spring. После выполнения настроек в Spring компоненты EJB могут внедряться в компоненты Spring, которые пользуются услугами EJB. Мы также увидели, как можно упростить объявление прокси-объектов для доступа к компонентам EJB с помощью элементов <jee:local-slsb> и <jee:remote-slsb> в Spring 2.0.

Затем мы повернулись в сторону разработки новых компонентов EJB. Даже при том, что Spring не имеет механизма для непосредственного размещения 2.x EJB в контейнере Spring, тем не менее в Spring имеется набор базовых классов, обладающих поддержкой Spring, на основе которых можно создавать новые компоненты EJB. Эти базовые классы экспортируют контекст приложения Spring в EJB, благодаря чему компонент EJB может делегировать выполнение операций объектам POJO, управляемым фреймворком Spring.

Наконец, мы познакомились с фреймворком Pitchfork, весьма интересным расширением Spring, позволяющим использовать аннотации EJB 3 для внедрения зависимостей и поддержки аспектов в контейнере Spring.

Вы уже наверняка заметили, что количество непрочитанных страниц в книге стремительно уменьшается. Наше путешествие по фреймворку Spring подходит к концу. Но, прежде чем закончить, мы сделаем еще несколько коротких остановок. В следующей главе мы исследуем некоторые особенности фреймворка Spring, которые не рассматривались ни в одной из предыдущих глав, включая организацию доступа к объектам в JNDI, отправку электронной почты и выполнение заданий по расписанию.



Глава 17. Прочее

В этой главе рассматриваются следующие темы:

- ❑ импортование внешних настроек;
- ❑ связывание ресурсов JNDI с компонентами Spring;
- ❑ отправка электронной почты;
- ❑ выполнение заданий по расписанию;
- ❑ асинхронные методы.

Я не знаю, как устроен ваш дом, но во многих домах (включая мой) имеется так называемая кладовка. В кладовке часто хранятся полезные и даже необходимые вещи. Вещи, такие как отвертки, шариковые ручки, скрепки для бумаг и запасные ключи. Не то, чтобы это было полное барахло, просто иногда для них не находится другого места.

К настоящему моменту мы перелопатили массу интересного материала и исследовали самые дальние закоулки фреймворка Spring. Для каждой большой темы была выделена отдельная глава. Но у нас осталось еще несколько интересных тем, которые мне хотелось бы раскрыть, но они не настолько большие, чтобы описывать их в отдельных главах.

Эта глава является своеобразной кладовкой. Но не думайте, что темы, затрагиваемые здесь, бесполезны. В этой главе вы найдете ценные советы. Здесь вы увидите, как импортировать внешние настройки в Spring, шифровать значения свойств, работать с объектами JNDI, отправлять электронную почту и подготавливать методы для выполнения в фоновом режиме, и все это с использованием фреймворка Spring.

Прежде всего посмотрим, как вынести настройку значений свойств из конфигурации Spring во внешние файлы определения свойств, которыми можно управлять без необходимости выполнять повторную сборку и развертывание приложения.

17.1. Импортирование внешних настроек

Часто полную конфигурацию приложения легко можно уместить в один файл. Но иногда может оказаться удобнее извлечь некоторые фрагменты конфигурации и поместить в отдельный файл с определением свойств. Например, во многих приложениях в конфигурационном файле определяется источник данных. В Spring настройка источника данных в конфигурационном файле могла бы иметь такой вид:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      p:driverClassName="org.hsqldb.jdbcDriver"
      p:url="jdbc:hsqldb:hsq://localhost/spitter/spitter"
      p:username="spitterAdmin"
      p:password="t0ps3cr3t" />
```

Как видите, вся информация, необходимая для *подключения к базе данных*, находится в объявлении компонента. Такое определение имеет две малоприятные особенности:

- ❑ если потребуется изменить URL базы данных, имя пользователя или пароль, придется редактировать файл конфигурации; затем пересобирать и повторно развертывать приложение;
- ❑ имя пользователя и пароль – это секретная информация, которая не должна попасть в чужие руки.

В ситуациях, подобных этой, было бы лучше не включать эти сведения непосредственно в конфигурационный файл Spring. Фреймворк Spring предусматривает два механизма импортирования настроек из файлов с определениями свойств, располагающихся за пределами развернутого приложения:

- ❑ *механизм подстановки переменных-заполнителей* в определениях значений свойств, выполняющий подстановку значений из внешнего файла;
- ❑ *механизм переопределения свойств*, замещающий значения свойств компонентов значениями из внешнего файла.

Кроме того, существует открытый проект Jasypt¹, предлагающий альтернативные реализации механизмов, перечисленных выше, ко-

¹ <http://www.jasypt.org>.



торые могут извлекать требуемые значения из зашифрованных файлов с определениями свойств.

Мы рассмотрим все эти механизмы по очереди и начнем с самого простого – механизма подстановки переменных-заполнителей.

17.1.1. Подстановка переменных-заполнителей

В версиях Spring ниже 2.5 для настройки механизма подстановки переменных-заполнителей в определении контекста Spring необходимо было объявить компонент `PropertyPlaceholderConfigurer`. Хотя это было совсем несложно, тем не менее начиная с версии Spring 2.5 эта процедура была упрощена еще больше добавлением нового элемента `<context:property-placeholder>` в конфигурационное пространство имен `context`. Теперь механизм подстановки переменных-заполнителей можно настраивать так:

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

Согласно этим настройкам, механизм подстановки переменных-заполнителей будет извлекать значения свойств из файла с именем `db.properties`, находящимся в корневом каталоге библиотеки классов (`classpath`). Но его точно так же можно было бы настроить на извлечение значений свойств из файла, находящегося в файловой системе:

```
<context:property-placeholder  
    location="file:///etc/db.properties" />
```

Что касается содержимого файла `db.properties`, он должен хранить (как минимум) значения свойств компонента `DataSource`:

```
jdbc.driverClassName=org.hsqldb.jdbcDriver  
jdbc.url=jdbc:hsqldb:hsqldb://localhost/spitter/spitter  
jdbc.username=spitterAdmin  
jdbc.password=t0ps3cr3t
```

Теперь можно заменить значения, объявленные в конфигурации Spring, переменными-заполнителями, опираясь на определения в файле `db.properties`:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      p:driverClassName="${jdbc.driverClassName}"
      p:url="${jdbc.url}"
      p:username="${jdbc.username}"
      p:password="${jdbc.password}" />
```

Возможно, кому-то важно будет знать, что область применения механизма подстановки переменных-заполнителей не ограничивается свойствами компонентов, определяемых в XML-файле. Его можно также использовать для настройки свойств, отмеченных аннотацией `@Value`. Например, если в программном коде имеется определение компонента, требующего настройки JDBC URL, в аннотации `@Value` можно использовать переменную-заполнитель `${jdbc.url}`, как показано ниже:

```
@Value("${jdbc.url}")
String databaseUrl;
```

Более того, переменные-заполнители можно использовать даже в самом файле с определениями свойств. Например, значение свойства `jdbc.url` можно определить с помощью переменных-заполнителей, чтобы разбить это определение на несколько частей:

```
jdbc.protocol=hsqldb:hsq
db.server=localhost
db.name=spitter
jdbc.url=jdbc:${jdbc.protocol}://${db.server}/${db.name}/${db.name}
```

Здесь я определил три свойства, `jdbc.protocol`, `db.server` и `db.name`. А также четвертое свойство, использующее три других для конструирования URL базы данных.

Выше были описаны основы механизма подстановки переменных-заполнителей в Spring. Но на этом его возможности не ограничиваются. В первую очередь посмотрим, как справиться с переменными-заполнителями в свойствах, определение которых отсутствует.

Подстановка отсутствующих значений

Что произойдет, если переменная-заполнитель не определена в файле свойств? Или, хуже того, атрибут `location` указывает на несуществующий файл свойств?



В подобных ситуациях во время загрузки контекста Spring и создания компонентов по умолчанию будет возбуждено исключение. Однако существует возможность избежать появления исключения, определив в элементе `<context:property-placeholder>` атрибуты `ignore-resource-not-found` и `ignore-unresolvable`:

```
<context:property-placeholder
    location="file:///etc/myconfig.properties"
    ignore-resource-not-found="true"
    ignore-unresolvable="true"
    properties-ref="defaultConfiguration"/>
```

Когда эти свойства установлены в значение `true`, механизм подстановки переменных-заполнителей не будет возбуждать исключение при встрече отсутствующей переменной-заполнителя или в отсутствие файла свойств и просто оставит переменные-заполнители как есть.

Но если переменные-заполнители не будут замещены какими-то значениями, разве это хорошо? В конце концов, значение `${jdbc.url}` не может использоваться для доступа к базе данных. Это недопустимый JDBC URL.

Было бы гораздо лучше иметь возможность внедрять значения по умолчанию вместо бесполезных переменных-заполнителей. Такую возможность обеспечивает атрибут `properties-ref`. В этом атрибуте определяется идентификатор компонента `java.util.Properties`, содержащего значения свойств по умолчанию. Ниже приводится элемент `<util:properties>`, определяющий значения по умолчанию для свойств, имеющих отношение к базе данных:

```
<util:properties id="defaultConfiguration">
    <prop key="jdbc.url">jdbc:mysql://localhost/spitter</prop>
    <prop key="jdbc.driverClassName">org.mysql.jdbc.Driver</prop>
    <prop key="jdbc.username">spitterAdmin</prop>
    <prop key="jdbc.password">t0ps3cr3t</prop>
</util:properties>
```

Если теперь какая-либо из переменных-заполнителей не будет найдена в файле `db.properties`, на ее место будет подставлено значение по умолчанию, объявленное в компоненте `defaultConfiguration`.

Определение значений переменных-заполнителей из системных свойств

К настоящему моменту мы видели, как определять значение переменных-заполнителей в файле свойств и в элементе `<util:properties>`.

Однако существует возможность получать значения этих переменных из системных свойств. Для этого достаточно определить атрибут system-properties-mode в компоненте <component:property-placeholder>. Например:

```
<context:property-placeholder
    location="file:///etc/myconfig.properties"
    ignore-resource-not-found="true"
    ignore-unresolvable="true"
    properties-ref="defaultConfiguration"
    system-properties-mode="OVERRIDE"/>
```

Значение OVERRIDE в атрибуте system-properties-mode указывает, что элемент <component:property-placeholder> должен отдавать предпочтение системным свойствам, а не определениям свойств в файле db.properties или в компоненте defaultConfiguration. Значение OVERRIDE – это одно из трех возможных значений в атрибуте system-properties-mode:

- FALLBACK – значения для переменных-заполнителей извлекаются из системных свойств, только если они не определены в файле свойств;
- NEVER – никогда не извлекать значения для переменных-заполнителей из системных свойств;
- OVERRIDE – отдавать предпочтение системным свойствам при разрешении переменных-заполнителей.

По умолчанию элемент <component:property-placeholder> будет пытаться получить значения переменных-заполнителей из файла свойств и в случае неудачи использовать системные свойства, как при значении FALLBACK в атрибуте system-properties-mode.

17.1.2. Переопределение свойств

Другой подход к импортированию внешних настроек, поддерживаемый фреймворком Spring, заключается в переопределении значений свойств компонентов с помощью файлов свойств. В этом случае переменные-заполнители не требуются. Вместо этого свойства компонентов либо связываются со значениями по умолчанию, либо вообще оставляются несвязанными. Если внешнее объявление свойства соответствует свойству компонента, для его инициализации используется внешнее значение.

Например, рассмотрим компонент dataSource, настраивавшийся выше с помощью механизма подстановки переменных-заполните-

лей. Его первоначальное объявление с жестко определенными значениями свойств выглядело, как показано ниже:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource"
      p:driverClassName="org.hsqldb.jdbcDriver"
      p:url="jdbc:hsqldb:hsq://localhost/spitter/spitter"
      p:username="spitterAdmin"
      p:password="t0ps3cr3t" />
```

В предыдущем разделе было показано, как объявлять значения по умолчанию с помощью элемента `<util:properties>` и механизма подстановки переменных-заполнителей. В отличие от него, механизм переопределения свойств позволяет оставить значения по умолчанию в свойствах, а об остальном он позаботится сам.

Настройка механизма переопределения свойств выполняется практически так же, как настройка механизма подстановки переменных-заполнителей. Разница лишь в том, что вместо элемента `<component:property-placeholder>` используется элемент `<component:property-override>`:

```
<context:property-override
    location="classpath:/db.properties" />
```

Чтобы механизм переопределения свойств смог сопоставить ключи в файле db.properties со свойствами компонентов в контексте приложения Spring, идентификаторы компонентов и имена свойств должны отображаться на ключи свойств в файле, как показано на рис. 17.1.

Как видите, ключ свойства во внешнем файле состоит из идентификатора компонента и имени свойства, разделяемых точкой. Ес-



Рис. 17.1. Механизм переопределения свойств определяет, какие свойства компонентов должны быть переопределены, отображая ключи в файле свойств на идентификаторы компонентов и имена их свойств

ли вернуться к началу раздела 17.1, можно заметить, что ключи в файле db.properties были достаточно близки к этому требованию, но не соответствовали ему полностью. Все ключи свойств начинались с префикса jdbc., вследствие чего они могли бы быть использованы, только если бы компоненту источника данных был присвоен идентификатор jdbc. Но он имеет идентификатор dataSource, поэтому необходимо внести некоторые изменения в файл db.properties:

```
dataSource.driverClassName=org.hsqldb.jdbcDriver
dataSource.url=jdbc:hsqldb:hsq://localhost/spitter/spitter
dataSource.username=spitterAdmin
dataSource.password=t0ps3cr3t
```

Теперь ключи в файле db.properties соответствуют свойствам компонента dataSource. В отсутствие файла db.properties будут задействованы значения свойств, явно указанные в конфигурационном файле. Но если файл db.properties присутствует и содержит только что представленные определения значений свойств, эти значения будут иметь более высокий приоритет, по сравнению со значениями в конфигурационном XML-файле.

Возможно, вам будет интересно узнать, что элемент <context:property-override> может настраиваться тем же набором атрибутов, что и элемент <context:property-placeholder>. Значения свойств по умолчанию можно точно так же настроить с помощью элемента <util:properties> и точно так же обеспечить получение значений из системных свойств.

К настоящему моменту вы познакомились с двумя механизмами импортирования значений свойств. Теперь вы легко сможете изменить URL базы данных или пароль, не пересобирая приложение и не разворачивая его повторно. Но осталось еще кое-что. Даже при том, что теперь пароль к базе данных отсутствует в определении контекста Spring, он все еще находится в открытом виде в файле свойств. Поэтому посмотрим, как использовать механизмы подстановки переменных-заполнителей и переопределения свойств из проекта Jasypt, чтобы зашифровать пароль, хранящийся во внешнем файле.

17.1.3. Шифрование внешних определений свойств

Проект Jasypt – это замечательная библиотека, упрощающая шифрование. Она обладает широкими возможностями, знакомство



с которыми выходит далеко за рамки этой книги. Но относительно темы импортирования внешних настроек компонентов библиотека Jasypt содержит особые реализации механизмов подстановки переменных-заполнителей и переопределения свойств, которые способны читать определения свойств из зашифрованных внешних файлов.

Как упоминалось выше, в версии Spring 2.5 появилось пространство имен context и его элементы `<context:property-placeholder>` и `<context:property-overrider>`. Прежде необходимо было настраивать `PropertyPlaceholderConfigurer` и `PropertyOverrideConfigurer` как компоненты, чтобы получить необходимые функциональные возможности.

Реализация механизмов подстановки переменных-заполнителей и переопределения свойств в библиотеке Jasypt в настоящее время не имеет собственного пространства имен. Поэтому оба механизма, как и в версиях Spring ниже 2.5, должны настраиваться в виде элементов `<bean>`.

Например, следующий элемент `<bean>` настраивает механизм подстановки переменных-заполнителей свойств:

```
<bean class=
    "org.jasypt.spring.properties.EncryptablePropertyPlaceholderConfigurer"
    p:location="file:///etc/db.properties">
    <constructor-arg ref="stringEncrypter" />
</bean>
```

А следующий элемент `<bean>` – механизм переопределения свойств:

```
<bean class=
    "org.jasypt.spring.properties.EncryptablePropertyOverrideConfigurer"
    p:location="file:///etc/db.properties">
    <constructor-arg ref="stringEncrypter" />
</bean>
```

Независимо от выбранного механизма следует настроить свойство `location`, определяющее местоположение файла свойств. И оба требуют указать в аргументе конструктора объект, реализующий шифрование.

В Jasypt существует понятие *строкового шифратора*, под которым подразумевается класс, определяющий стратегию шифрования строковых значений. Строковый шифратор используется механизмами подстановки/переопределения для шифрования/десифрования значений, обнаруживаемых во внешних файлах свойств. Для наших

целей отлично подойдет класс `StandardPBESStringEncryptor`, входящий в состав библиотеки Jasypt:

```
<bean id="stringEncrypter"
      class="org.jasypt.encryption.pbe.StandardPBESStringEncryptor"
      p:config-ref="environmentConfig" />
```

Единственное, что необходимо классу `StandardPBESStringEncryptor` для выполнения своей работы, – это алгоритм и пароль, использовавшийся при шифровании данных. Если заглянуть в описание класса `StandardPBESStringEncryptor`, можно увидеть, что он имеет свойства `algorithm` и `password`. Эти свойства можно было бы определить непосредственно в объявлении компонента `stringEncryptor`.

Но если оставить пароль шифрования в конфигурации Spring, действительно ли доступ к базе данных останется защищенным? Фигурально выражаясь, это все равно, что спрятать ключи от базы данных в сейфе и оставить ключи от сейфа на столе рядом с ним. В лучшем случае мы немного осложним доступ к базе данных, но не обезопасим его.

Вместо настройки пароля непосредственно в конфигурации Spring я настроил свойство `config` класса `StandardPBESStringEncryptor`, определив в нем ссылку на компонент `EnvironmentStringPBEConfig`. `EnvironmentStringPBEConfig` позволяет настраивать параметры шифрования, такие как пароль, в переменных окружения. `EnvironmentStringPBEConfig` – это обычный компонент, объявленный, как показано ниже:

```
<bean id="environmentConfig" class=
      "org.jasypt.encryption.pbe.config.EnvironmentStringPBEConfig"
      p:algorithm="PBEWithMD5AndDES"
      p:passwordEnvName="DB_ENCRYPTION_PWD" />
```

Я не против того, чтобы определять алгоритм в конфигурации Spring – здесь я указал его как `PBEWithMD5AndDES`. А вот пароль, согласно этим настройкам, будет храниться за пределами Spring, в переменной окружения с именем `DB_ENCRYPTION_PWD`.

У кого-то может появиться вопрос, как перемещение пароля в переменную окружения повысит безопасность. Разве злоумышленник не сможет прочитать переменную окружения так же легко, как конфигурационный файл Spring? Да, это так. Но суть в том, что эта переменная окружения, как предполагается, будет устанавливаться администратором системы непосредственно перед запуском при-



ложения и стираться сразу после этого. К тому моменту свойства компонента источника данных уже будут установлены, и надобность в переменной окружении отпадет.

Механизмы импортирования значений свойств компонентов из внешних источников – это лишь один из способов управления настройками, которые желательно держать в тайне и/или необходимо изменять после развертывания приложения. Еще один способ решения этих проблем – импортирование целых объектов из JNDI и настройка Spring на извлечение этих объектов в контекст Spring. Этот способ рассматривается в следующем разделе.

17.2. Внедрение объектов из JNDI

Интерфейс доступа к службам и каталогов (Java Naming and Directory Interface, JNDI) – это Java API, позволяющий отыскивать объекты по их именам в каталоге (часто, но не обязательно, в каталоге LDAP). Механизм JNDI предоставляет Java-приложениям доступ к центральному репозиторию, позволяя сохранять и извлекать прикладные объекты. В приложениях Java EE механизм JNDI обычно используется для хранения и извлечения источников данных JDBC и диспетчеров транзакций JTA. Кроме того, компоненты сеансов, определяемые спецификацией EJB 3, также часто находят себе пристанище в JNDI.

Но если некоторые из наших прикладных объектов хранятся в JNDI, за пределами Spring, как тогда внедрить их в объекты, управляемые фреймворком Spring?

В этом разделе мы познакомимся с поддержкой JNDI в Spring – упрощенным уровнем абстракции над стандартным JNDI API. Абстракция JNDI в Spring позволяет определять в конфигурационном файле Spring информацию, необходимую для поиска объектов в JNDI. После этого вы сможете внедрять объекты из JNDI в свойства других компонентов Spring, как если бы они были обычными компонентами в контексте приложения Spring.

Чтобы получить более полное представление о возможностях абстракции JNDI в Spring, попробуем отыскать объект в репозитории JNDI без помощи Spring.

17.2.1. Работа с обычным JNDI API

Реализация поиска объектов в репозитории JNDI может оказаться весьма утомительным занятием. Например, допустим, что нам

требуется выполнить типичную операцию по извлечению объекта `javax.sql.DataSource` из JNDI. Используя только JNDI API, реализовать эту операцию можно было бы, как показано ниже:

```
InitialContext ctx = null;
try {
    ctx = new InitialContext();

    DataSource ds =
        (DataSource) ctx.lookup("java:comp/env/jdbc/SpitterDatasource");
} catch (NamingException ne) {
    // обработка исключений ...
} finally {
    if(ctx != null) {
        try {
            ctx.close();
        } catch (NamingException ne) {}
    }
}
```

Если прежде вам приходилось заниматься реализацией поиска объектов в JNDI, вы, возможно, сможете понять, что делает этот фрагмент. Возможно, вам приходилось писать нечто подобное, чтобы извлечь объект из репозитория JNDI. Прежде чем повторить попытку, разберем подробнее, что, собственно, требуется сделать.

- ❑ Только чтобы отыскать объект `DataSource`, необходимо создать и закрыть начальный контекст. Может показаться, что это требует написать не так много программного кода, но этот шаблонный код не связан непосредственно с преследуемой целью – извлечением объекта источника данных.
- ❑ Необходимо перехватить или, по крайней мере, повторно возбудить исключение `javax.naming.NamingException`. Если вы предпочтете перехватить его, необходимо также предусмотреть соответствующую обработку. Если вы предпочтете повторно возбудить его, обработку вынужден будет выполнить вызывающий программный код. Так или иначе, но где-то в приложении должна быть предусмотрена обработка этого исключения.
- ❑ Программный код оказывается *тесно связанным* с репозиторием JNDI. Программе нужен всего лишь объект `DataSource`. И не важно, откуда он будет получен, из JNDI или откуда-то еще. Но если программа содержит программный код, подобный этому, она вынуждена будет извлекать `DataSource` из JNDI.

- Программный код оказывается тесно связанным с определенным именем JNDI – в данном случае с `java:comp/env/jdbc/SpitterDatasource`. Конечно, имя можно было бы определить в файле свойств, но тогда вам придется добавить еще больше шаблонного кода, чтобы получить имя JNDI из файла.

После внимательного изучения становится очевидным, что большая часть программного кода – это шаблонный код, реализующий поиск в JNDI, который остается практически неизменным от приложения к приложению. Единственная строка, имеющая непосредственное отношение к извлечению источника данных:

```
DataSource ds =
    (DataSource) ctx.lookup("java:comp/env/jdbc/SpitterDatasource");
```

Еще большее беспокойство, чем шаблонный код для работы с JNDI, доставляет тот факт, что при таком подходе приложение точно знает, откуда извлекается объект источника данных. Он всегда извлекается из репозитория JNDI. Как показано на рис. 17.2, объект DAO, использующий объект источника данных, оказывается прочно привязанным к JNDI. Это обстоятельство делает практически невозможным использование данного программного кода в окружении, где репозиторий JNDI недоступен или его использование нежелательно.



Рис. 17.2. Применение обычного JNDI API для извлечения зависимостей приводит к образованию тесной связи с JNDI, что осложняет использование программного кода, где JNDI недоступен

Например, представьте, что реализация поиска источника данных включена в класс, подвергающийся модульному тестированию. В идеале объект должен тестироваться в изолированной среде, независимо от каких-либо других объектов. Хотя класс и не зависит от источника данных, извлекаемого из JNDI, он зависит от самого механизма JNDI. То есть модульный тест также оказывается завися-

щим от JNDI, и на момент тестирования сервер JNDI должен быть доступен.

Однако все вышесказанное не отменяет того факта, что иногда действительно бывает необходимо иметь возможность извлекать объекты из JNDI. Источники данных часто настраиваются на сервере приложений, чтобы воспользоваться преимуществом организации пулов соединений на сервере, которые затем извлекаются прикладным программным кодом для доступа к базе данных. Но как организовать извлечение объектов из JNDI, не попадая в зависимость от JNDI?

Ответ можно найти в механизме внедрения зависимостей. Вместо того чтобы пытаться получить источник данных из JNDI, программный код должен просто принимать источник данных, который может быть получен откуда угодно, – программный код должен определять свойство `DataSource`, куда будет выполняться внедрение. Откуда будет получен объект для класса, использующего его, не имеет никакого значения.

Итак, источник данных находится в репозитории JNDI. Так как же настроить Spring, чтобы внедрить объект, хранящийся в JNDI?

17.2.2. Внедрение объектов из JNDI

Конфигурационное пространство имен jee в Spring обеспечивает возможность работы с JNDI способом, не создающим тесной зависимости. Внутри этого пространства имен имеется элемент `<jee:jndi-lookup>`, который упрощает внедрение объектов из JNDI в Spring.

Для иллюстрации вернемся к примеру из главы 6. Там элемент `<jee:jndi-lookup>` использовался для извлечения объекта `DataSource` из JNDI:

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true" />
```

Атрибут `jndi-name` определяет имя объекта в репозитории JNDI. Это имя используется по умолчанию для поиска объекта в JNDI. Но если поиск выполняется в контейнере Java EE, тогда может потребоваться добавить префикс `java:comp/env/`. Этот префикс можно добавить вручную, при определении значения в атрибуте `jndi-name`. Однако если указать в атрибуте `resource-ref` значение `true`, то же самое будет выполнено элементом `<jee:jndi-lookup>` автоматически.

Объявив компонент dataSource, его можно внедрить в свойство dataSource. Например, его можно использовать для настройки фабрики сеансов Hibernate:

```
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.annotation.
      ↳ AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    ...
</bean>
```

Как показано на рис. 17.3, когда фреймворк Spring будет связывать компонент sessionFactory, он внедрит объект DataSource, полученный из JNDI, в свойство dataSource фабрики сеансов.

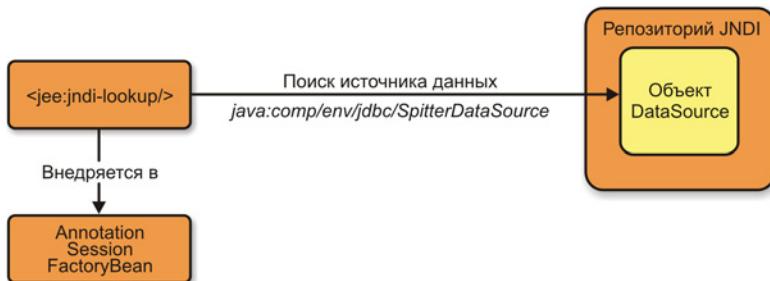


Рис. 17.3. Элемент `<jee:jndi-lookup>` извлекает объект из JNDI, превращая его в компонент в контексте приложения Spring.
После этого его можно внедрить в любой другой объект

Вся прелесть использования элемента `<jee:jndi-lookup>` для извлечения объекта из JNDI – в том, что единственным фрагментом кода, который знает, что объект DataSource извлекается из JNDI, является объявление компонента dataSource в XML-файле. Компонент sessionFactory понятия не имеет (и его это не интересует), откуда получен объект DataSource. А это означает, что если вы решите получить объект DataSource из драйвера JDBC, достаточно будет просто переопределить компонент dataSource, указав класс DriverManagerDataSource.

Теперь источник данных будет извлекаться из репозитория JNDI и внедряться в фабрику сеансов. Мы избавились от реализации поиска объекта в JNDI! Всякий раз, когда нам потребуется источник данных, мы сможем получить его в виде компонента dataSource из контекста приложения Spring.

Как было показано выше, внедрение компонентов, хранящихся в JNDI, в Spring реализуется очень просто. А теперь познакомимся с несколькими способами влиять на то, когда и как объект должен извлекаться из JNDI, начав с кеширования.

Кеширование объектов из JNDI

Часто объекты, извлекаемые из репозитория JNDI, используются многократно. Источник данных, например, будет необходим всякий раз, когда потребуется выполнить обращение к базе данных. Было бы слишком расточительно извлекать источник данных из JNDI каждый раз, когда появится нужда в нем. По этой причине элемент `<jee:jndi-lookup>` по умолчанию кеширует объекты, полученные из JNDI.

Кеширование отлично подходит для большинства ситуаций. Но препятствует «горячей» реорганизации объектов в JNDI. Если вам понадобится изменить объект в JNDI, это повлечет за собой необходимость перезапустить приложение Spring, чтобы оно извлекло новый объект.

Если предполагается, что извлекаемый приложением объект будет часто изменяться, кеширование желательно отключить в элементе `<jee:jndi-lookup>`. Для этого нужно указать значение `false` в атрибуте `cache`:

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true"
    cache="false"
    proxy-interface="javax.sql.DataSource" />
```

Значение `false` в атрибуте `cache` сообщает элементу `<jee:jndi-lookup>`, что объект всегда должен извлекаться из репозитория JNDI. Обратите внимание, что при этом также был определен атрибут `proxy-interface`. Поскольку объект в JNDI может измениться в любой момент, элемент `<jee:jndi-lookup>` не имеет никакой возможности узнать фактический тип объекта. Атрибут `proxy-interface` указывает ожидаемый тип извлекаемого объекта.

Отложенная загрузка объектов из JNDI

Иногда приложению не требуется извлекать объект из JNDI немедленно. Например, представьте, что объект из JNDI используется только при определенных условиях. В этой ситуации может ока-



затьсяя нежелательным загружать объект до того момента, когда он действительно будет необходим.

По умолчанию элемент `<jee:jndi-lookup>` извлекает объекты из JNDI в момент запуска контекста приложения. Однако имеется возможность отсрочить загрузку объекта до момента, когда он станет необходим, определив в атрибуте `lookup-on-startup` значение `false`:

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true"
    lookup-on-startup="false"
    proxy-interface="javax.sql.DataSource" />
```

Как и в случае с атрибутом `cache`, при установке значения `false` в атрибуте `lookup-on-startup` необходимо определить атрибут `proxy-interface`. Это обусловлено тем, что элемент `<jee:jndi-lookup>` не знает тип извлекаемого объекта, пока действительно не извлечет его. Атрибут `proxy-interface` сообщает ему ожидаемый тип объекта.

Запасные объекты на случай неудачи

Теперь вы знаете, как внедрять объекты из репозитория JNDI. Жизнь прекрасна. Но что, если объект не будет найден в JNDI?

Например, приложение может рассчитывать получить источник данных из JNDI при выполнении в промышленном окружении. Но на этапе разработки это может оказаться невозможным. Если приложение будет настроено на получение источника данных из репозитория JNDI на этапе эксплуатации, то на этапе разработки загрузка объекта будет терпеть неудачу. Как тогда обеспечить извлечение компонента источника данных из JNDI на этапе эксплуатации и явно настраивать его на этапе разработки?

Как было показано выше, элемент `<jee:jndi-lookup>` прекрасно справляется с извлечением объектов из JNDI и их внедрением в контекст приложения Spring. Но он также имеет аварийный механизм на случай, когда запрашиваемый объект отсутствует в репозитории JNDI. Чтобы задействовать его, необходимо настроить атрибут `default-ref`.

Например, допустим, что источник данных в конфигурации Spring объявляется как компонент класса `DriverManagerDataSource`:

```
<bean id="devDataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource"
```

```
lazy-init="true">
<property name="driverClassName"
          value="org.hsqldb.jdbcDriver" />
<property name="url"
          value="jdbc:hsqldb:hsq://localhost/spitter/spitter" />
<property name="username" value="sa" />
<property name="password" value="" />
</bean>
```

Этот источник данных будет использоваться на этапе разработки. Но на этапе разработки должен использоваться источник данных из JNDI, настраиваемый системным администратором. В этом случае элемент `<jee:jndi-lookup>` мог бы быть настроен, как показано ниже:

```
<jee:jndi-lookup id="dataSource"
    jndi-name="/jdbc/SpitterDS"
    resource-ref="true"
    default-ref="devDataSource" />
```

Здесь в атрибут `default-ref` записывается ссылка на компонент `devDataSource`. Если элемент `<jee:jndi-lookup>` не сможет получить из JNDI объект с именем `/jdbc/SpitterDS`, он будет использовать компонент `devDataSource`. А поскольку компонент источника данных по умолчанию содержит атрибут `lazy-init` со значением `true`, он не будет создан, пока действительно не понадобится.

Как видите, элемент `<jee:jndi-lookup>` существенно упрощает внедрение объектов из JNDI в контекст приложения Spring. Однако, как оказывается, элемент `<jee:jndi-lookup>` также может использоваться для внедрения компонентов сеансов EJB. Посмотрим, как это сделать.

17.2.3. Внедрение компонентов EJB в Spring

Компоненты сеансов, определяемые спецификацией EJB 3, – это всего лишь объекты, хранящиеся в репозитории JNDI, подобно любым другим объектам в JNDI. Поэтому элемент `<jee:jndi-lookup>` можно с успехом использовать для получения компонентов сеанса EJB 3. Но как быть, если потребуется внедрить в контекст приложения компонент сеанса EJB 2?

Чтобы получить доступ к компоненту сеанса EJB 2, сначала нужно извлечь объект из JNDI. Но это будет объект, реализующий собственный (домашний) интерфейс EJB, а не сам компонент EJB.

Чтобы получить ссылку на EJB, необходимо вызвать метод `create()` домашнего интерфейса.

К счастью, вам не придется вникать во все эти детали при организации доступа к компонентам сеансов EJB 2 с помощью фреймворка Spring. На этот случай вместо элемента `<jee:jndi-lookup>` фреймворк Spring предлагает два других элемента из пространства имен `jee`:

- ❑ `<jee:local-slsb>` – для доступа к локальным компонентам сеансов;
- ❑ `<jee:remote-slsb>` – для доступа к удаленным компонентам сеансов.

Оба элемента действуют подобно элементу `<jee:jndi-lookup>`. Например, объявить ссылку на удаленный компонент сеанса в Spring с помощью `<jee:remote-slsb>` можно следующим образом:

```
<jee:remote-slsb id="myEJB"
    jndi-name="my.ejb"
    business-interface="com.habuma.ejb.MyEJB" />
```

Атрибут `jndi-name` – это имя в JNDI, используемое для поиска домашнего интерфейса компонента EJB. А атрибут `business-interface` определяет прикладной интерфейс, реализуемый компонентом EJB. Определив ссылку на компонент EJB, как показано выше, компонент `myEJB` можно внедрять в свойства любых других компонентов, имеющие тип `com.habuma.ejb.MyEJB`.

Аналогично, с помощью элемента `<jee:local-slsb>`, объявляется ссылка на локальный компонент:

```
<jee:local-slsb id="myEJB"
    jndi-name="my.ejb"
    business-interface="com.habuma.ejb.MyEJB" />
```

Выше мы обсудили особенности использования элементов `<jee:local-slsb>` и `<jee:remote-slsb>` для объявления компонентов сеанса EJB 2 в Spring. Но самое интересное в этих элементах состоит в том, что они могут применяться для внедрения компонентов сеансов EJB 3. Они способны извлекать объекты из JNDI и автоматически определять, являются ли они реализациями домашнего интерфейса EJB 2 или компонентами сеанса EJB 3. При получении реализации домашнего интерфейса EJB 2 они автоматически вызывают метод `create()`. В противном случае они предполагают, что имеют дело с компонентом EJB 3, и внедряют его в контекст Spring.

Поддержка извлечения объектов из JNDI может пригодиться, чтобы организовать обращение к объектам, настраиваемым за пределами фреймворка Spring. Как было показано выше, источники данных могут настраиваться на сервере приложений и предоставляться приложениям посредством JNDI. И, как будет показано ниже, поддержка JNDI в Spring может также пригодиться для реализации отправки электронной почты. Познакомимся с абстракцией электронной почты в Spring.

17.3. Отправка электронной почты

В главе 13 мы использовали поддержку обмена сообщениями в Spring для асинхронной отправки другим пользователям приложения Spitter извещений о появлении новых сообщений. Теперь можно воспользоваться поддержкой электронной почты в Spring для отправки электронных писем.

В состав фреймворка Spring входит API абстракции электронной почты, упрощающий отправку электронных писем.

17.3.1. Настройка отправки электронной почты

Основу абстракции электронной почты в Spring составляет интерфейс MailSender. Как следует из его имени и показано на рис. 17.4, реализация MailSender отправляет электронную почту. Фреймворк Spring уже имеет одну готовую реализацию интерфейса MailSender – класс JavaMailSenderImpl.



Рис. 17.4. Интерфейс MailSender является основой абстракции электронной почты в Spring. Он обеспечивает отправку электронной почты на почтовый сервер для дальнейшей доставки

А как насчет CosMailSenderImpl? Старые версии Spring, вплоть до версии Spring 2.0 (включительно), предоставляли еще одну реализацию интерфейса MailSender – класс CosMailSenderImpl. Эта реализация была убрана в версии Spring 2.5. Если вы все еще используете ее, перед миграцией на версию Spring 2.5 или Spring 3.0 необходимо перейти на использование класса JavaMailSenderImpl.



Чтобы воспользоваться реализацией `JavaMailSenderImpl`, в контексте приложения Spring необходимо объявить компонент:

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl"
      p:host="${mailserver.host}" />
```

Свойство `host` определяет доменное имя почтового сервера, используемого для отправки электронной почты. Здесь это имя настраивается с помощью механизма подстановки переменных-заполнителей, чтобы иметь возможность изменять настройки почтового сервера за пределами Spring. По умолчанию реализация `JavaMailSenderImpl` предполагает, что почтовый сервер принимает соединения на порту 25 (стандартный SMTP-порт). Если ваш почтовый сервер использует другой порт, укажите правильное значение в свойстве `port`. Например:

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl"
      p:host="${mailserver.host}"
      p:port="${mailserver.port}" />
```

Аналогично, если почтовый сервер требует аутентификации, потребуется также определить значения свойств `username` и `password`:

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl"
      p:host="${mailserver.host}"
      p:port="${mailserver.port}"
      p:username="${mailserver.username}"
      p:password="${mailserver.password}" />
```

Последнее определение демонстрирует полный комплекс параметров настройки, необходимых для доступа к почтовому серверу. При желании можно воспользоваться готовым сеансом связи с почтовым сервером, предварительно настроенным в JNDI. Посмотрим, как настроить `JavaMailSenderImpl` на использование такого готового сеанса.

Использование готового сеанса, находящегося в JNDI

Возможно, в вашем репозитории JNDI уже имеется настроенный компонент `javax.mail.MailSession` (например, сохраненный сервером

приложений). В этом случае компонент JavaMailSenderImpl может использовать уже готовый сеанс MailSession.

Мы уже знаем, как с помощью элемента <jee:jndi-lookup> извлекать объекты из JNDI. Поэтому воспользуемся им для загрузки настроенного объекта сеанса из JNDI:

```
<jee:jndi-lookup id="mailSession"
    jndi-name="mail/Session" resource-ref="true" />
```

Теперь готовый сеанс можно внедрить в компонент mailSender:

```
<bean id="mailSender"
    class="org.springframework.mail.javamail.JavaMailSenderImpl"
    p:session-ref="mailSession" />
```

Внедрив готовый объект сеанса в свойство session компонента JavaMailSenderImpl, мы заменили явно указанное имя сервера (а также имя пользователя и пароль). Теперь сеанс связи с почтовым сервером можно хранить и настраивать в JNDI. Реализация JavaMailSenderImpl отвечает за отправку электронной почты, но она не может взаимодействовать с почтовым сервером по своей инициативе.

Внедрение компонента отправки почты в компонент службы

После настройки компонента, занимающегося отправкой электронной почты, его необходимо внедрить в компонент, который будет использовать его. Наиболее подходящим местом для отправки почты в приложении Spitter является класс SpitterEmailServiceImpl. Этот класс имеет свойство mailSender, отмеченное аннотацией @Autowired:

```
@Autowired
JavaMailSender mailSender;
```

Когда фреймворк Spring создаст компонент SpitterEmailServiceImpl, он попытается отыскать компонент, реализующий интерфейс MailSender, который можно было бы внедрить в свойство mailSender. В результате он найдет компонент mailSender и внедрит его. Организовав внедрение компонента mailSender, можно приступать к созданию и отправке электронных писем.



17.3.2. Создание электронных писем

По условиям задачи нам требуется организовать отправку пользователям приложения Spitter электронных писем, чтобы известить их о появлении новых сообщений, поэтому нам необходим некоторый метод, который будет получать адрес электронной почты и объект Spittle и отправлять электронное письмо. Таким методом будет sendSimpleSpittleEmail(), представленный в листинге 17.1.

Листинг 17.1. Отправка электронной почты с использованием реализации MailSender

```
public void sendSimpleSpittleEmail(String to, Spittle spittle) {  
    SimpleMailMessage message = new SimpleMailMessage(); // Создать письмо  
  
    String spitterName = spittle.getSpitter().getFullName();  
    message.setFrom("noreply@spitter.com"); // Обратный адрес  
    message.setTo(to);  
    message.setSubject("New spittle from " + spitterName);  
  
    message.setText(spitterName + " says: " +  
        spittle.getText());  
  
    mailSender.send(message); // Отправить  
}
```

Первое, что делает метод sendSimpleSpittleEmail(), – конструирует экземпляр класса SimpleMailMessage. Это объект электронного письма, как следует из его имени, прекрасно подходящий для отправки деловых сообщений.

Далее он определяет параметры сообщения. Адреса отправителя и получателя определяются с помощью методов setFrom() и setTo() объекта письма. После определения темы вызовом метода setSubject() виртуальный «конверт» готов к отправке. Остается только вызвать метод setText(), чтобы добавить текстовое содержимое.

В заключение выполняется отправка письма вызовом метода send() компонента, отвечающего за отправку.

Простые электронные письма великолепно подходят на начальном этапе. Но как быть, если позднее потребуется добавлять в них вложения? Или что, если у вас появится желание придать тексту письма более привлекательное оформление? Посмотрим, как с помощью Spring можно принарядить наши электронные письма, начав с добавления вложений.

Добавление вложений

Вся хитрость отправки электронных писем с *вложениями* заключается в создании составных сообщений – электронных писем, содержащих несколько частей, одна из которых играет роль тела письма, а остальные являются вложениями.

Класс `SimpleMailMessage` слишком... как бы это сказать... слишком прост, чтобы с его помощью отправлять вложения. Чтобы отправить составное электронное письмо, необходимо создать *MIME-сообщение* (*Multipurpose Internet Mail Extensions* – многоцелевые расширения электронной почты Интернета). Для этого можно воспользоваться методом `createMimeMessage()` объекта, реализующего отправку электронной почты:

```
MimeMessage message = mailSender.createMimeMessage();
```

Поехали! Теперь у нас есть заготовка MIME-сообщения. На первый взгляд может показаться, что достаточно просто определить адреса отправителя и получателя, тему сообщения, добавить некоторый текст и вложения. В принципе, так оно и есть, но не все так просто. Прикладной интерфейс класса `javax.mail.internet.MimeMessage` слишком сложен, чтобы его можно было использовать без постоянной помощи. К счастью, в Spring имеется класс `MimeMessageHelper`, оказывающий такую помощь.

Чтобы воспользоваться услугами класса `MimeMessageHelper`, необходимо создать его экземпляр, передав его конструктору объект `MimeMessage`:

```
MimeMessageHelper helper = new MimeMessageHelper(message, true);
```

Значение `true`, передаваемое конструктору во втором параметре, сообщает ему, что это сообщение является составным сообщением.

Получив экземпляр `MimeMessageHelper`, можно приступать к сборке электронного письма. Единственное существенное отличие состоит в том, что конструирование выполняется с помощью метода вспомогательного объекта, а не самого сообщения:

```
String spittleName = spittle.getSpitter().getFullName();
helper.setFrom("noreply@spitter.com");
helper.setTo(to);
helper.setSubject("New spittle from " + spittleName);
helper.setText(spittleName + " says: " + spittle.getText());
```

Единственное, что осталось сделать перед отправкой письма, – вложить изображение купона. Для этого необходимо загрузить изображение как ресурс и передать его методу addAttachment() вспомогательного объекта:

```
FileSystemResource couponImage =
    new FileSystemResource("/collateral/coupon.png");
helper.addAttachment("Coupon.png", couponImage);
```

Здесь сначала выполняется загрузка файла coupon.png из библиотеки классов приложения (classpath) с помощью объекта FileSystemResource, а затем вызывается метод addAttachment(). В первом параметре методу передается имя вложения, а во втором – ресурс с изображением.

Составное электронное письмо сконструировано. Теперь его можно отправить. Полная версия метода sendSpittleEmailWithAttachment() представлена в листинге 17.2.

Листинг 17.2. Класс MimeMessageHelper упрощает создание электронных писем с вложениями

```
public void sendSpittleEmailWithAttachment(
    String to, Spittle spittle) throws MessagingException {
    MimeMessage message = mailSender.createMimeMessage();
    MimeMessageHelper helper =
        new MimeMessageHelper(message, true);           // Создание письма

    String spitterName = spittle.getSpitter().getFullName();
    helper.setFrom("noreply@spitter.com");
    helper.setTo(to);
    helper.setSubject("New spittle from " + spitterName);

    helper.setText(spitterName + " says: " + spittle.getText());

    FileSystemResource couponImage =
        new FileSystemResource("/collateral/coupon.png");
    helper.addAttachment("Coupon.png", couponImage);      // Вложение

    mailSender.send(message);
}
```

Добавление вложений в электронные письма – это лишь одна из возможностей, поддерживаемых составными сообщениями. Помимо этого, они позволяют определять тело письма в формате HTML,

имеющем более привлекательное оформление, чем простой текст. Посмотрим, как отправлять письма с текстом в формате HTML с помощью объекта `MimeMessageHelper`.

Отправка электронных писем с текстом в формате HTML

Отправка электронных писем с содержимым в формате HTML практически не отличается от отправки писем с содержимым в простом текстовом формате. Для этого необходимо лишь записать текст письма в формате HTML, просто передав строку с разметкой HTML методу `setText()` вспомогательного объекта и значения `true` во втором параметре:

```
helper.setText("<html><body><img src='cid:spitterLogo'>" +  
    "<h4>" + spittle.getSpitter().getFullName() + " says...</h4>" +  
    "<i>" + spittle.getText() + "</i>" +  
    "</body></html>", true);
```

Второй параметр указывает, что текст, передаваемый в первом параметре, является разметкой HTML, чтобы для данной части письма был установлен соответствующий тип содержимого.

Обратите внимание, что разметка HTML в данном примере содержит тег ``, отображающий логотип приложения Spitter в тексте письма. Чтобы получить логотип из Всемирной паутины, в атрибуте `src` можно указать полный URL `http:`. Но в данном случае в текст письма внедряется сам логотип. Значение `cid:spitterLogo` указывает, что изображение содержится в одной из частей письма, с идентификатором `spitterLogo`.

Добавление встроенного изображения в сообщение во многом напоминает добавление вложения. Только вместо метода `addAttachment()` вспомогательного объекта вызывается метод `addInline()`:

```
ClassPathResource image = new ClassPathResource("spitter_logo_50.png");  
helper.addInline("spitterLogo", image);
```

Первый параметр метода `addInline()` определяет идентификатор встроенного изображения, совпадающий с идентификатором, указанным в атрибуте `src` тега ``. Во втором параметре передается ссылка на ресурс с изображением, созданным здесь с помощью объекта `ClassPathResource`, который извлекает изображение из библиотеки классов приложения.

Кроме немного отличающегося вызова метода `setText()` и использования метода `addInline()`, отправка писем с текстом в формате HTML ничем не отличается от отправки простых текстовых писем с вложениями. Для сравнения ниже представлена новая версия метода `sendRichSpitterEmail()`.

```
public void sendRichSpitterEmail(String to, Spittle spittle)
    throws MessagingException {
    MimeMessage message = mailSender.createMimeMessage();
    MimeMessageHelper helper = new MimeMessageHelper(message, true);
    helper.setFrom("noreply@spitter.com");
    helper.setTo("craig@habuma.com");
    helper.setSubject("New spittle from " +
        spittle.getSpitter().getFullName());
    helper.setText("<html><body><img src='cid:spitterLogo'>" + // HTML-текст
        "<h4>" + spittle.getSpitter().getFullName() + " says...</h4>" +
        "<i>" + spittle.getText() + "</i>" +
        "</body></html>", true);

    ClassPathResource image = new ClassPathResource("spitter_logo_50.png");
    helper.addInline("spitterLogo", image);      // Встроенное изображение
    mailSender.send(message);
}
```

Теперь у вас есть возможность отправлять электронные письма с текстом в формате HTML и встроенными изображениями! На этом можно остановиться и объявить о завершении реализации отправки электронных писем. Но мне кажется ошибкой составление тела письма с применением операции конкатенации строк. Прежде чем оставить тему электронной почты, посмотрим, как избавиться от операций конкатенации строк с помощью шаблонов сообщений.

Создание шаблонов электронных писем

Проблема конструирования электронных писем с применением операций конкатенации строк состоит в том, что этот прием не позволяет увидеть, как будет выглядеть письмо. Не каждому под силу осмыслить разметку HTML и представить, как она будет отображаться. Кроме того, само внедрение разметки HTML в исходные тексты на языке Java является сложной задачей, и было бы здорово, если можно было бы переместить оформление письма в шаблон, который может воспроизводиться редактором с графическим интерфейсом.

Нам необходим способ, позволяющий выразить форматирование письма в виде HTML-шаблона и затем преобразовать этот шаблон в строку для передачи методу `setText()` вспомогательного объекта. Когда речь заходит о преобразовании шаблонов в строки, в числе одних из лучших инструментов называют процессор шаблонов Apache Velocity¹.

Чтобы задействовать процессор Velocity для форматирования электронных писем, сначала необходимо внедрить объект `VelocityEngine` в компонент `SpitterEmailServiceImpl`. Для этой цели в Spring имеется удобный фабричный компонент `VelocityEngineFactoryBean`, создающий объекты `VelocityEngine` в контексте приложения Spring. Объявление `VelocityEngineFactoryBean` выглядит, как показано ниже:

```
<bean id="velocityEngine"
      class="org.springframework.ui.velocity.VelocityEngineFactoryBean">
    <property name="velocityProperties">
      <value>
        resource.loader=class
        class.resource.loader.class=
          org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader
      </value>
    </property>
</bean>
```

Единственное свойство, требующее настройки в компоненте `VelocityEngineFactoryBean`, – это свойство `velocityProperties`. В данном случае он настраивается на загрузку шаблонов Velocity из библиотеки классов (за более подробной информацией о настройке процессора Velocity обращайтесь к документации).

Теперь процессор Velocity можно внедрить в `SpitterEmailServiceImpl`. Поскольку компонент `SpitterEmailServiceImpl` автоматически регистрируется сканером компонентов, для автоматического связывания его свойства `velocityEngine` можно воспользоваться аннотацией `@Autowired`:

```
@Autowired
VelocityEngine velocityEngine;
```

После настройки свойства `velocityEngine` с его помощью можно выполнить преобразование шаблона Velocity в строку для отправки

¹ <http://velocity.apache.org>.

в виде электронного письма. Чтобы помочь в этом, в Spring имеется класс `VelocityEngineUtils`, упрощающий слияние шаблона Velocity и дополнительных данных в строку. Ниже показано, как пользоваться этим классом:

```
Map<String, String> model = new HashMap<String, String>();
model.put("spitterName", spitterName);
model.put("spittleText", spittle.getText());
String emailText = VelocityEngineUtils.mergeTemplateToString(
    velocityEngine, "emailTemplate.vm", model );
```

Подготовка к обработке шаблона начинается с создания отображения (`Map`) для хранения данных модели, используемой шаблоном. В предыдущей версии, использующей конкатенацию строк, нам необходимы были полное имя пользователя и текст его сообщения, следовательно, эта же информация потребуется и здесь. После этого, чтобы сконструировать текст электронного письма, достаточно просто вызвать метод `mergeTemplateToString()` класса `VelocityEngineUtils`, передав ему процессор Velocity, путь к шаблону (относительно корня библиотеки классов) и отображение модели.

Все, что осталось сделать, – передать получившийся текст письма методу `setText()` вспомогательного объекта:

```
helper.setText(emailText, true);
```

Сам шаблон хранится в корневом каталоге библиотеки классов (`classpath`), в файле с именем `emailTemplate.vm`, содержащем следующие строки:

```
<html>
<body>
    <img src='cid:spitterLogo'>
    <h4>${spitterName} says...</h4>
    <i>${spittleText}</i>
</body>
</html>
```

Как видите, содержимое файла шаблона читается намного проще, чем его версия в программном коде на Java. Следовательно, его проще будет сопровождать и изменять. На рис. 17.5 показано, как выглядит электронное письмо, созданное с его помощью.



Рис. 17.5. С помощью шаблона Velocity и нескольких встроенных изображений можно придать скучному письму довольно привлекательный внешний вид

Взглянув на рис. 17.5, я увидел массу возможностей улучшить шаблон, чтобы придать электронному письму более привлекательный вид. Но оставляю это вам в качестве самостоятельного упражнения.

У нас осталась еще одна весьма привлекательная особенность фреймворка Spring, которую я оставил на сладкое! Посмотрим, как с помощью Spring организовать выполнение коротких заданий в фоновом режиме.

17.4. Выполнение заданий по расписанию и в фоновом режиме

Наиболее заметными функциональными особенностями большинства приложений являются те, что реагируют на действия пользователя. Пользователь заполняет форму, щелкает на кнопке, и приложение реагирует, обрабатывая информацию, сохраняя ее в базе данных и производя некоторый вывод.

Но иногда приложения выполняют свою собственную работу, без участия пользователя. В то время как пользователь щелкает на кнопки, приложение может выполнять фоновые задания, не требующие участия пользователя.



Фреймворк Spring поддерживает две разновидности фоновых заданий:

- выполнение заданий по расписанию;
- асинхронные методы.

Задания, выполняемые по расписанию, включают операции, которые требуется выполнять регулярно, через определенные интервалы времени или в определенное время суток. Асинхронные методы, наоборот, вызываются приложением, но они немедленно возвращают управление вызывающей программе, продолжая выполнение в фоновом режиме.

Независимо от выбранного типа заданий для их поддержки необходимо включить всего одну строку в конфигурационный файл контекста приложения Spring:

```
<task:annotation-driven/>
```

Элемент `<task:annotation-driven/>` включает в Spring автоматическую поддержку заданий, выполняемых по расписанию, и *асинхронных методов*. Эти методы идентифицируются с помощью аннотаций `@Scheduled` и `@Async` соответственно.

Посмотрим, как пользоваться этими аннотациями, начав с аннотации `@Scheduled`, применяемой для организации вызова методов по расписанию.

17.4.1. Объявление методов, вызываемых по расписанию

Имеющие опыт работы с фреймворком Spring знают, что он поддерживает возможность вызова методов по расписанию. Но до недавнего времени для этого требовалось включать дополнительные настройки в конфигурационный файл Spring. Во втором издании этой книги я на 10 страницах описывал, как реализовать вызовы методов по расписанию.

С появлением новой аннотации `@Scheduled` в Spring 3 все изменилось. Если раньше для достижения цели требовалось несколько строк в XML-файле и некоторое количество компонентов, то теперь достаточно одного элемента `<task :annotation-driven>` и единственной аннотации. И для описания их использования мне потребуется явно меньше 10 страниц.

Чтобы превратить метод в задание, выполняемое по расписанию, его необходимо отметить аннотацией `@Scheduled`. Например, чтобы

обеспечить автоматический вызов метода через каждые 24 часа (86 400 000 миллисекунд):

```
@Scheduled(fixedRate=86400000)
public void archiveOldSpittles() {
    // ...
}
```

Атрибут `fixedRate` определяет периодичность вызова метода в миллисекундах. В данном случае указано, что вызовы должны осуществляться через каждые 86 400 000 миллисекунд. Определить интервал времени, который должен пройти между моментом завершения метода и моментом следующего его вызова, можно с помощью атрибута `fixedDelay`:

```
@Scheduled(fixedDelay=86400000)
public void archiveOldSpittles() {
    // ...
}
```

Выполнение заданий через определенные интервалы времени – довольно удобная возможность. Но иногда может потребоваться возможность более точного управления временем вызова метода. Атрибуты `fixedRate` и `fixedDelay` позволяют управлять только периодичностью вызова метода, но не моментом, когда это произойдет. Определить точные моменты вызова метода можно с помощью атрибута `cron`:

```
@Scheduled(cron="0 0 0 * * SAT")
public void archiveOldSpittles() {
    // ...
}
```

Значение в атрибуте `cron` указывается в формате планировщика Cron. Для тех, кто не знаком с планировщиком Cron, разберем, как определяется значение в атрибуте `cron`. Это значение состоит из шести элементов (возможно, из семи), разделенных пробелами. В порядке следования слева направо эти элементы имеют следующие значения:

1. Секунды (0–59).
2. Минуты (0–59).



3. Часы (0–23).
4. День месяца (1–31).
5. Месяц (1–12 или JAN–DEC).
6. День недели (1–7 или SUN–SAT).
7. Год (1970–2099).

Каждый из этих элементов может определяться как конкретное значение (6), как диапазон (9–12), как список (9, 11, 13) или как шаблонный символ (например, *). День месяца и день недели являются взаимоисключающими, поэтому необходимо явно определить, какой из них не будет использоваться, указав в ненужном элементе знак вопроса (?). В табл. 17.1 приводится список некоторых выражений в стиле планировщика Cron, которые можно использовать в атрибуте cron.

В примере выше указано, что архивирование старых сообщений должно выполняться каждую субботу в полночь. Но благодаря использованию выражений в стиле планировщика Cron богатство возможных вариантов практически не ограничено. Атрибуты fixedRate и fixedDelay ограничивают наши возможности фиксированными периодами времени, тогда как атрибут cron позволяет запланировать вызов метода в любые моменты времени. Я уверен, что вы сможете придумать массу интересных выражений в стиле Cron для определения расписаний вызовов своих методов.

Таблица 17.1. Некоторые примеры выражений в стиле планировщика Cron

Выражение Cron	Описание
0 0 10, 14, 16 * * ?	Ежедневно, в 10:00, 14:00 и в 16:00
0 0, 15, 30, 45 * 1-30 * ?	Каждые 15 минут с 1 по 30 число месяца
30 0 0 1 1 ? 2012	Через 30 секунд после полуночи 1 января 2012 года
0 0 8-17 ? * MON-FRI	Каждый час в течение рабочего дня с понедельника по пятницу

17.4.2. Объявление асинхронных методов

Производительность приложения может оцениваться как фактическая и воспринимаемая человеком. Безусловно, *фактическая производительность* приложения (определяется временем, необходимым на выполнение операций) играет важную роль. Но даже когда фактическая производительность далека от идеала, этот недостаток отчасти можно замаскировать за воспринимаемой производительностью.

Воспринимаемая производительность – это именно то, что означает ее название. Кому интересно знать, как долго выполняется та или иная операция, если по мнению пользователя приложение откликается на его действия немедленно? Например, допустим, что добавление нового сообщения является достаточно продолжительной операцией. При синхронном ее выполнении воспринимаемая производительность будет прямо пропорциональна фактической производительности. Пользователь будет вынужден ждать, пока приложение будет сохранено на сервере.

Но если появится возможность выполнять метод `saveSpittle()` класса `SpitterService` асинхронно, тогда приложение сможет представить пользователю новую страницу, пока логика сохранения будет выполняться в фоновом режиме. Именно эту цель преследует аннотация `@Async`.

`@Async` – это простая аннотация, не имеющая атрибутов. Достаточно просто отметить метод компонента этой аннотацией, и он превратится в асинхронный метод. Сложно придумать что-либо более простое.

Например, ниже показано, как можно превратить метод `saveSpittle()` класса `SpittleServiceImpl` в асинхронный метод:

```
@Async  
public void addSpittle(Spittle spittle) {  
    ...  
}
```

Этого действительно достаточно. Когда приложение вызовет метод `saveSpittle()`, управление немедленно будет возвращено вызывающей программе. При этом сам метод `saveSpittle()` продолжит выполнение в фоновом режиме.

У вас может возникнуть вопрос: а что, если асинхронный метод возвращает некоторое значение? Если управление будет возвращено немедленно, как тогда он сможет вернуть результаты?

Так как асинхронные методы в Spring основаны на API параллельного выполнения, они могут возвращать объект, реализующий интерфейс `java.util.concurrent.Future`. Этот интерфейс представляет хранилище для некоторого значения, которое будет доступно спустя некоторое время после возврата из метода, но необязательно в точке вызова этого метода. В состав Spring входит класс `AsyncResult`, реализующий интерфейс `Future` и упрощающий работу со значениями, которые будут доступны в будущем.

Например, допустим, что имеется асинхронный метод, выполняющий сложные и длительные вычисления. Этот метод желательно было бы выполнять в фоновом режиме, но иметь возможность получать результаты сразу по его завершении. В подобном случае метод можно было объявить, как показано ниже:

```
@Async  
public Future<Long> performSomeReallyHairyMath(long input) {  
    // ...  
    return new AsyncResult<Long>(result);  
}
```

Пока этот метод выполняет длительные вычисления, вызывающая программа может продолжать заниматься другими делами. В результате вызова метода программа получит объект `Future` (фактически экземпляр класса `AsyncResult`), куда будет записан результат по окончании вычислений.

Когда результат будет готов, вызывающая программа сможет извлечь его вызовом метода `get()` объекта `Future`. А до тех пор программа может проверять готовность результата вызовами методов `isDone()` и `isCancelled()`.

17.5. В заключение

В этой главе мы познакомились с некоторыми темами, не имеющими прямого отношения ни к одной из глав.

Сначала мы познакомились с возможностью импортирования значений свойств компонентов, предоставляемой механизмами подстановки переменных-заполнителей и переопределения свойств. Мы также узнали, что внешние определения свойств можно не только импортировать, но и шифровать их, чтобы скрыть секретные сведения от постороннего взгляда.

Затем мы сделали еще один шаг и поместили в репозиторий JNDI целые объекты, а потом настроили Spring так, чтобы обеспечить извлечение этих объектов в контекст приложения и их внедрение в другие компоненты, как если бы они были обычными локальными компонентами.

Далее мы увидели, как с помощью Spring отправлять электронную почту. Абстракцию электронной почты в Spring нельзя назвать одной из самых захватывающих, однако она существенно упроща-

ет реализацию отправки электронной почты. Мы видели, как отправлять простые электронные письма, письма с текстом в формате HTML и письма с вложениями и встроенным содержимым.

В заключение мы коснулись темы выполнения фоновых заданий. Мы начали со знакомства с аннотациями, позволяющими организовать вызов методов по определенному расписанию. А затем – с аннотациями, превращающими обычные методы в асинхронные и позволяющими увеличить воспринимаемую производительность приложения.

17.6. Конец?

Как бы ни было грустно это признать, но мы добрались до конца книги. Однако это не говорит о том, что мы узнали все о фреймворке Spring. Как отмечалось в предисловии, я мог бы писать о фреймворке Spring *без остановки*. Но тогда эта книга никогда не попала бы к вам в руки, и я никогда не познал бы удовлетворения от того, что проспал всю ночь.

Несмотря на то что иногда приходилось принимать очень жесткие решения, касающиеся содержимого этой книги, я полагаю, что в ней удалось охватить наиболее важные темы, касающиеся разработки приложений на основе фреймворка Spring. И теперь вы обладаете всеми необходимыми знаниями, чтобы продолжить исследование других тем самостоятельно.

Итак, хотя эта глава завершает книгу «Spring в действии», ваше путешествие по фреймворку Spring только начинается. Я советую вам использовать полученные здесь знания в изучении других сторон фреймворка Spring, таких как Spring Integration, Spring Batch, Spring Dynamic Modules и (моего любимого) Spring Roo. К счастью, издательство Manning выпустило книги из серии «... в действии» для каждой из этих тем, которые помогут вам в дальнейших исследованиях:

- ❑ «Spring Integration in Action», Марк Фишер (Mark Fisher), Йонас Партинер (Jonas Partner), Мариус Богоевич (Marius Boeveici) и Ивайн Фулд (Iwein Fuld);
- ❑ «Spring Batch in Action», Тьерри Темплиер (Thierry Templier) и Арно Коголенес (Arnaud Cogoluègnes);
- ❑ «Spring Dynamic Modules in Action», Арно Коголенес (Arnaud Cogoluègnes), Тьерри Темплиер (Thierry Templier) и Энди Пайпер (Andy Piper);

- «Roo in Action», Гордон Диккенс (Gordon Dickens) и Кен Римпл (Ken Rimple).

И вы всегда сможете посетить форум <http://forum.springframework.org>, чтобы поближе познакомиться с этими и другими проектами, развивающимися в рамках фреймворка Spring.

Я получил массу удовольствия, надеюсь, что и у вас остались приятные впечатления.

Предметный указатель

Символы

@AfterReturning, аннотация, 225
@AfterThrowing, аннотация, 226
@Annotation(), указатель для точки внедрения, 208
@args(), указатель для точки внедрения, 208
@Aspect, аннотация, 225
@AspectJ, 224
 параметры, 228
 аннотация, 72
@Async, аннотация, 712, 715
@Autowired, аннотация, 65, 177, 183, 184, 272, 486, 703
 @Inject, аннотация, 184
 @Qualifier, аннотация, 180
 required, атрибут, 180
 и ключевое слово private, 179
 и конструкторы, 178
 и методы записи, 178
 необязательное автоматическое связывание, 179
 неоднозначные зависимости, 180
 создание собственных квалифицированных, 181
@Bean, аннотация, 194
@Before, аннотация, 225
@Cacheable, аннотация, 292
@CacheFlush, аннотация, 292
@Component, аннотация, 65, 189, 331, 581
@Configurable, аннотация, 134
@Configuration, аннотация, 193
@Controller, аннотация, 189, 331, 346
@CookieValue, аннотация, 66
@DeclareParents, аннотация, 230
 defaultImpl, атрибут, 230
 value, атрибут, 230
@Entity, аннотация, 270
@Inject, аннотация, 177, 184, 331
@ManagedAttribute, аннотация, 598
@ManagedOperation, аннотация, 598
@ManagedResource, аннотация, 598

@MappedSuperclass, аннотация, 270
@MessageDriven, аннотация, 576
@Named, аннотация, 185
@PathVariable, аннотация, 355, 511, 517, 518
@Pattern, аннотация, 357
@Pointcut, аннотация, 225
@PostAuthorize, аннотация, 466
@PostConstruct, аннотация, 65
@PostFilter, 466
@PostFilter, аннотация, 466
@PreAuthorize, аннотация, 466
@PreDestroy, аннотация, 65
@PreFilter, аннотация, 466
@Qualifier, аннотация, 65, 180
@Repository, аннотация, 189, 272, 281
@RequestBody, аннотация, 512, 531
@RequestHeader, аннотация, 66
@RequestMapping, аннотация, 329, 346, 351, 513, 517, 518, 521, 522, 529
@RequestParam, аннотация, 346, 363
@Resource, аннотация, 65, 177
@ResponseBody, аннотация, 512, 528, 529, 538
@ResponseStatus, аннотация, 521
@RolesAllowed, аннотация, 464, 465
@Scheduled, аннотация, 712
@Secured, аннотация, 464
@Service, аннотация, 189
@target(), указатель для точки внедрения, 208
@Transactional, аннотация, 281, 309, 319
@Valid, аннотация, 354
@Value, аннотация, 186, 365, 685
@WebMethod, аннотация, 501
@WebService, аннотация, 500
@within(), указатель для точки внедрения, 208
<action-state>, элемент, 400
<amq:connectionFactory>, элемент, 564
<amq:queue>, элемент, 565
<amq:topic>, элемент, 565

<aop:advisor>, элемент, 211
 <aop:after-returning>, элемент, 211, 214
 <aop:after-throwing>, элемент, 211, 214
 <aop:after>, элемент, 46, 211
 <aop:around>, элемент, 211
 <aop:aspectj-autoproxy/>, элемент, 226
 <aop:aspectj-autoproxy>, элемент, 211
 <aop:aspect>, элемент, 45, 211, 214, 215
 <aop:before>, элемент, 45, 211, 214
 <aop:config>, элемент, 216
 <aop:config>, элемент, 212, 213
 <aop:declare-parents>, элемент, 212, 223, 230
 <aop:pointcut>, элемент, 212, 215
 <aop:spring-configured>, элемент, 134
 <authentication-manager>, элемент, 454
 <beans>, компонент
 default-destroy-method, атрибут, 83
 default-init-method, атрибут, 83
 <beans>, элемент, 71
 default-autowire, атрибут, 175
 <bean>, компонент
 destroy-method, атрибут, 82
 init-method, атрибут, 82
 <bean>, элемент, 71
 abstract, атрибут, 118
 class, атрибут, 74
 dataSource, свойство, 303
 factory-method, атрибут, 79, 80
 id, атрибут, 73
 parent, атрибут, 118
 primary, атрибут, 172
 scope, атрибут, 80
 sessionFactory, свойство, 303
 transactionManagerName, свойство, 306
 автоматическое определение, 188
 автоматическое связывание, 177
 внутренние компоненты, 89
 внутри элемента <list>, 94
 внутри элементов <list>, 94
 пространство имен р, 91
 <c:forEach>, элемент, 349
 <cache>, элемент, 288
 <component:property-override>, элемент, 688
 <component:property-placeholder>, элемент, 687
 <constructor-arg>, элемент, 75, 565
 ref, атрибут, 78
 value, атрибут, 78
 избавление с помощью
 автоматического связывания
 через конструктор, 173
 <context:annotation-config>, элемент, 177
 <context:component-scan>, элемент, 188, 272, 331
 base-package, атрибут, 189
 фильтрация, 191
 <context:exclude-filter>, элемент, 191
 <context:include-filter>, элемент, 191
 <context:mbean-export>, элемент, 597, 600
 <context:mbean-server>, элемент, 591
 <context:property-placeholder>, элемент, 684
 <decision-state>, элемент, 400, 415
 <defaultCache>, элемент, 288
 <definition>, элемент, 351
 <ehcache:caching>, элемент, 290
 <ehcache:flushing>, элемент, 290
 <ehcache:proxy>, элемент, 289
 <end-state>, элемент, 401
 <entry>, элемент
 key-ref, атрибут, 97
 key, атрибут, 97, 591
 value-ref, атрибут, 97
 value, атрибут, 97
 <evaluate>, элемент, 400, 405, 411, 417
 <filter-name>, элемент, 438
 <filter>, элемент, 437
 <flow:flow-executor>, элемент, 394
 <flow:flow-location-pattern>, элемент, 395

- <flow:flow-registry>, элемент, 395
<flow>, элемент, 411
<form-login>, элемент, 440
<form>, элемент, 551
<global-method-security>, элемент, 463
<global-transitions>, элемент, 404
<http-basic>, элемент, 443
<http>, элемент, 438
 path-type, атрибут, 443
 use-expressions, атрибут, 444
<if>, элемент, 401
<input>, элемент, 362, 401, 411, 422
<intercept-url>, элемент, 438, 443
 pattern, атрибут, 438
<jdbc-user-service>, элемент, 455
<jee:jndi-lookup>, элемент, 249, 695
 jndi-name, атрибут, 250
 resource-ref, атрибут, 250
 кэширование, 697
<jee:jndilookup>, элемент, 279
<jee:local-slrb>, элемент, 666, 700
<jee:remote-slrb>, элемент, 666, 700
<jms:listener-container>, элемент, 578
<jms:listener>, элемент, 579
<lang:bsh>, элемент, 162
<lang:groovy>, элемент, 161
<lang:inline-script>, элемент, 165
<lang:jruby>, элемент, 160
<lang:property>, элемент, 163
<ldap-authentication-provider>, элемент, 458
<ldap-server>, элемент, 460
<listener>, элемент, 343
<list>, элемент, 92, 93
 внутри других элементов
 <list>, 94
 значения, 94
<logout>, элемент, 443
<lookup-method>, элемент, 131
<map>, элемент, 93, 97
<mvc:annotation-driven>, элемент, 329
<mvc:resources>, элемент, 326
 mapping, атрибут, 326
<null/>, элемент
 внедрение, 99
 внутри элементов <list>, 94
 и автоматическое связывание, 176
<output>, элемент, 411, 420
<password-compare>, элемент, 459
<pointcut>, элемент, 46
<property>, элемент, 101, 175
 ref, атрибут, 102
 value, атрибут, 86
<props>, элемент, 93, 98
<prop>, элемент, 98
<qualifier>, элемент, 181
<remember-me>, элемент, 462
<replaced-method>, элемент, 128
<s:url>, элемент, 341
<secured>, элемент, 427
<security:authentication>, элемент, 448
<security:authorize>, элемент, 449
<servlet-mapping>, элемент, 496, 644
<servlet-name>, элемент, 325
<servlet>, элемент, 324
<set>, элемент, 92, 93, 405
<sf:checkbox>, элемент, 353
<sf:errors>, элемент, 358
<sf:form>, элемент, 353, 362, 550
<sf:input>, элемент, 353
<sf:password>, элемент, 353
<subflow-state>, элемент, 401, 411
<task:annotation-driven/>, элемент, 712
<transition>, элемент, 403
<tx:advice>, элемент, 316
<tx:annotation-driven>, элемент, 319
 transactionmanager, атрибут, 319
<tx:attributes>, элемент, 316
<tx:method>, элемент, 316
 name, атрибут, 316
<user-service>, элемент, 453
<user>, элемент, 453
<util:properties>, элемент, 686, 689
<value>, элемент, внутри элементов <list>, 94
<var>, элемент, 405
<view-state>, элемент, 399
<wsdl:operation>, элемент, 643
<wsdl:portType>, элемент, 643

**A**

Абстрактные компоненты, 119, 122, 123
 общие свойства, 122, 123
 переопределение наследуемых свойств, 121
 Автоматическое определение, 188
 Автоматическое связывание, 169
 @Autowired, аннотация, 178
 @Inject, аннотация, 184
 @Qualifier, аннотация, 180
 конечных точек JAX-WS, 500
 необязательное, 179
 неоднозначные зависимости, 180
 свойств, 169, 177
 autodetect, 170, 174
 byName, 170
 byType, 170, 171
 constructor, 170, 173
 по умолчанию, 174
 со значением null, 176
 смещивание автоматического и явного связывания, 175
 с помощью аннотаций, 177
 Агент MBean, 590
 Аннотации
 @AfterReturning, 225
 @AfterThrowing, 226
 @Aspect, 225
 @AspectJ, 72, 224
 @Async, 712, 715
 @Autowired, 65, 177, 183, 184, 272, 486, 703
 @Bean, 194
 @Before, 225
 @Cacheable, 292
 @CacheFlush, 292
 @Component, 65, 189, 331, 581
 @Configurable, 134
 @Configuration, 193
 @Controller, 189, 331, 346
 @CookieValue, 66
 @DeclareParents, 230
 @Entity, 270
 @Inject, 177, 184, 331
 @ManagedAttribute, 598

@ManagedOperation, 598
 @ManagedResource, 598
 @MappedSuperclass, 270
 @MessageDriven, 576
 @Named, 185
 @PathVariable, 355, 511, 517, 518
 @Pattern, 357
 @Pointcut, 225
 @PostAuthorize, 466
 @PostConstruct, 65
 @PostFilter, 466
 @PreAuthorize, 466
 @PreDestroy, 65
 @PreFilter, 466
 @Qualifier, 65, 180
 @Repository, 189, 272, 281
 @RequestBody, 512, 531
 @RequestHeader, 66
 @RequestMapping, 329, 346, 351, 513, 517, 518, 521, 522, 529
 @RequestParam, 346, 363
 @Resource, 65, 177
 @ResponseBody, 512, 528, 529, 538
 @ResponseStatus, 521
 @RolesAllowed, 464, 465
 @Scheduled, 712
 @Secured, 464
 @Service, 189
 @Transactional, 281, 309, 319
 @Valid, 354
 @Value, 186, 365, 685
 @WebMethod, 501
 @WebService, 500
 AnnotationSessionFactoryBean, 270
 JSR-303, 66
 внедрение ресурсов с помощью Pitchfork, 676
 и Spring MVC, 328
 и автоматическое определение, 189
 и внедрение, 230
 и выражения на языке SpEL, 186
 и связывание, 177
 объявление перехватчиков с помощью Pitchfork, 678
 объявление управляемых компонентов MBean, 597
 определение транзакций с помощью аннотаций, 318

- поддерживаемые фреймворком Pitchfork, 675
поддержка автоматического связывания, 177
Арбитры представлений, 334
Асинхронные взаимодействия, 556
Lingo, библиотека, 583
гарантированная доставка, 562
независимость от местоположения, 562
отсутствие ожидания, 561
преимущества, 561
Асинхронные методы, 712, 714
Аспектно-ориентированное программирование
внедрение, 221
Аспектно-ориентированное программирование, 40, 46
аспекты, 201, 203
внедрение, 203
вплетение, 203
защита методов, 463
и Spring Security, 433
методы как точки сопряжения, 207
модуль Spring AOP, 57
объявление аспектов, 224
определение, 199
пример, 43, 46
советы, 202
срезы множества точек сопряжения, 203
точки сопряжения, 202
фреймворки, 205
Аспекты, 40, 46, 201, 203
 <aop:advisor>, элемент, 211
 <aop:after-returning>, элемент, 211
 <aop:after-throwing>, элемент, 211
 <aop:after>, элемент, 211
 <aop:around>, элемент, 211
 <aop:aspectj-autoproxy/>, элемент, 226
 <aop:aspectj-autoproxy>, элемент, 211
 <aop:aspect>, элемент, 211
 <aop:before>, элемент, 211
 <aop:config>, элемент, 212
 <aop:declare-parents>, элемент, 212, 223
 <aop:pointcut>, элемент, 212
AspectJ, внедрение, 231
внедрение, 203, 221
вплетение, 203
в сравнении с наследованием и делегированием, 201
как обертки, 42
определение, 201
последующая операция (after advice), 46
предварительная операция (before advice), 46
пример, 43, 46
сквозные задачи, 41
советы, 202
срезы множества точек сопряжения, 203
точки сопряжения, 202
Атрибуты
 abstract, 118
 access, 445, 450
 arg-names, 220
 authentication-failure-url, 440
 authorities, 453
 authorities-by-username-query, 457
 auto-config, 439
 base-package, 189
 base-path, 395
 business-interface, 700
 cache, 697
 class, 74
 connectionFactory, 579
 content, 352
 cron, 713
 data-source, 455
 default-autowire, 175
 default-destroy-method, 83
 default-impl, 223
 defaultImpl, 230
 default-init-method, 83
 default-ref, 698
 delegate-ref, 223
 destroy-method, 82
 else, 401
 expression, 191, 400



factory-method, 79, 80, 234
fixedDelay, 713
fixedRate, 713
group-search-base, 458
group-search-filter, 458
hash, 459
headers, 529
id, 73, 91
ignore-resource-not-found, 686
ignore-unresolvable, 686
implement-interface, 223
init-method, 54, 82
jndi-name, 250, 695, 700
jsr250-annotations, 465
key, 97, 591
key-ref, 97
lazy-init, 665, 699
ldif, 461
login, 440
login-processing-url, 440
logout-url, 443
lookup-on-startup, 698
mapping, 326
method, 521
methodName, 290
model, 399
name, 316, 416, 453
objectName, 598
on, 403
on-exception, 404
params, 351
parent, 118
password, 453
password-attribute, 459
path, 353, 396
path-type, 443
pattern, 438, 443
physicalName, 566
pointcut, 215
pointcut-ref, 216
pre-post-annotations, 466
primary, 172
properties-ref, 686
property, 448
proxy-interface, 697
ref, 78, 102, 214
refresh-check-delay, 164
registration, 600
required, 180
requires-channel, 446
resource-ref, 250, 695
root, 460
scope, 80, 449
script-interfaces, 160
script-source, 160
start-state, 411
system-properties-mode, 687
then, 401
to, 403
transactionmanager, 319
type, 191
types-matching, 223
url, 460
use-expressions, 444
userPassword, 459
user-search-base, 458
user-search-filter, 458
user-service, 454
value, 78, 86, 97, 101, 230, 522
value-ref, 97
var, 448
view, 399, 402
when, 291
определение с помощью
интерфейсов, 596
управляемые, 590
Атрибуты транзакций, 309
время ожидания, 315
правила отмены, 315
правила распространения, 310
только для чтения, 314
уровни изоляции, 312
Аутентификация, 448, 452
с использованием LDAP, 457
с использованием базы
данных, 455
с использованием репозитория
в памяти, 453
средствами протокола HTTP, 442
Б
Базы данных, 238
аутентификация, 455
извлечение строк
с использованием JDBC, 256

- обновление строк
 - с использованием JDBC, 255
 - подключение, 683
 - Безопасность
 - веб-последовательностей, 427
 - методов, 463
 - минимальная настройка, 436
 - на уровне представлений, 447
 - Борьба с разбуханием кода, 566
 - Брокер сообщений
 - ActiveMQ, 563
 - настройка, 563
 - Брокеры объектных запросов, 481
 - Брокеры сообщений, 558
- ## B
- Веб-службы
 - «contract-first», модель разработки, 613, 614, 616
 - обзор, 613, 616
 - определение, 614, 616
 - «contract-last», модель разработки, 613
 - обработка сообщений, 623
 - отправка сообщений, 651
 - связывание, 632
 - шаблоны, 648
 - шлюзы, 656
 - Вложения, 705
 - Внедрение, 203, 221
 - <aop:declare-parents>, элемент, 223
 - аспектов AspectJ, 231
 - внутренних компонентов, 89
 - в свойства компонентов, 84
 - значений, 86
 - и аннотации, 230
 - коллекций, 92, 97
 - пустых значений, 99
 - ссылок через конструкторы, 76
 - через конструкторы, 74
 - через методы доступа, 85
 - через методы доступа компонентов, 84
 - Внедрение зависимостей, 35, 50
 - @Resource, аннотация, 65
 - и JNDI, 695
- ## Г
- Глобальные переходы, 404
- ## Д
- Данные в последовательностях, 404



- область видимости диалога, 406
 область видимости запроса, 406
 область видимости кадра, 406
 область видимости последовательности, 406
 область видимости представления, 406
 сбор данных, 412
 Даты, форматирование в Velocity, 372
 Декларативное кэширование с помощью аннотаций, 292
 Декларативное управление транзакциями, 309
 Делегирование, в сравнении с аспектами, 201
 Динамические компоненты MBean, 588
 Диспетчеры сущностей управляемые контейнером, 275
 управляемые приложением, 274
 Диспетчеры транзакций, 301
 Доменные объекты, 132
 Доступ к данным JDBC, 252, 265
 иерархия исключений, 241, 243
 источники данных, 252
 кэширование, 282
 решения, 284
 настройка источников данных, 249
 обзор, 239
 обратные вызовы, 245
 разбухание кода, 254
 с использованием SimpleJdbcTemplate, 259
 слои, 239, 243
 шаблоны, 244, 245, 246
 Дочерние компоненты объявление, 118
 абстрактных компонентов, 119, 122, 123
 общих абстрактных свойств, 122, 123
- 3**
 Завершение сеанса работы, 443
 Запасные объекты, на случай неудачи, 698

- Запросы HttpServletRequest, 436
 безопасность, 436
 определение типа возвращаемых данных, 525
 прием объектов в ответах, 543
 принудительное использование протокола HTTPS, 446
 путь в Spring MVC, 322
 Запросы к последовательности, 396
 Значения внедрение, 86
 литералы, 100

И

- Идемпотентность, 520
 Извещения, 606
 прием, 609
 Именованные параметры, 261
 Импортирование внешних настроек, 683
 Инициализация компонентов, 82, 83
 Инициализация по требованию владельца, 79
 Интернационализация, 147
 Интерфейсы ApplicationContext, 207
 publishEvent(), метод, 151
 ApplicationContextAware, 53, 153
 ApplicationContextAware-Processor, 143
 BeanFactoryAware, 53, 153
 BeanNameAware, 53, 154
 BeanPostProcessor, 54, 141, 144
 ContextLoaderListener, 343
 DataSource, 259, 303
 DisposableBean, 54, 83
 EntityManager, 275
 EntityManagerFactory, 274, 277, 279, 305
 FactoryBean, 269
 Filter, 437
 Future, 715
 InitializingBean, 54, 83
 java.beans.PropertyEditor, 136
 java.util.Collection, 92, 95
 java.util.Map, 93, 96, 263

- java.util.Properties, 93, 98
java.util.Set, 95
JpaDialect, 305
MailSender, 701, 703
MailSession, 703
Marshaller, 636
MessageCreator, 572
MessageDrivenBean, 576
MessageListener, 576
MessageSource, 148
MultipartResolver, 367
NotificationListener, 609
NotificationPublisherAware, 607
org.springframework.context.
ApplicationListener, 152
PersistenceProvider, 275, 277
Provider, 185
Serializable, 480
Session, 268
SessionFactory, 272
Transaction, 304
TransactionCallback, 307
TransactionDefinition, 310
TransactionManager, 306
Unmarshaller, 636
UserTransaction, 306
и слабая связность, 88
определение экспортимемых
операций и атрибутов, 596
скрытие слоя хранения
данных, 240
- Исключения**
catch, блоки, 241, 244
DataAccessException, 243
Hibernate, 242
IllegalArgumentException, 209
ImageUploadException, 365
InstanceAlreadyExists-
Exception, 600
IOException, 534
JmsException, 569
JMSEException, 568
MalformedURLExceptions, 481
NamingException, 693
NoSuchBeanDefinition-
Exception, 180
NotModifiedException, 540
- NullPointerException, 104, 176
RemoteAccessException, 480
RemoteException, 479, 481
RestClientException, 538
SpitterClientException, 534
SQLException, 48, 241, 255
UnmarshallingFailure-
Exception, 641
UnsupportedOperation-
Exception, 471
URISyntaxException, 541
ValidationFailureException, 641
иерархия исключений JDBC
в сравнении с иерархией
исключений Spring, 242
иерархия исключений доступа
к данным, 241
- Источники данных**
BasicDataSource, класс, 250
JDBC, 249, 252
JNDI, 249
настройка, 252
пулы соединений, 250
- K**
- Каскадирование, 266
Квалификаторы
пользовательские, 181
Классы
AbstractDom4jPayloadEndpoint,
624
AbstractDomPayloadEndpoint, 624
AbstractJDomPayloadEndpoint,
624
AbstractJmsMessageDrivenBean,
670
AbstractMarshallingPayloadEnd-
point, 624
AbstractMessageDrivenBean, 670
AbstractSaxPayloadEndpoint, 624
AbstractStatefulSessionBean, 670
AbstractStatelessSessionBean, 670
AbstractStaxEventPayloadEnd-
point, 624
AbstractStaxStreamPayloadEnd-
point, 624
AbstractXomPayloadEndpoint, 624



AnnotationAwareAspectJAuto-
ProxyCreator, 226
AnnotationSessionFactoryBean,
класс, 270
AsyncResult, 715
BurlapServiceExporter, 492
CcDaoSupport, класс, 248
CommonsHttpMessageSender, 650
CommonsMultipartResolver, 367
ConnectorServerFactoryBean, 602
CosMailSenderImpl, 701
DefaultMethodSecurityExpression
Handler, 471
DelegatingFilterProxy, 437
DriverManagerDataSource,
класс, 252
DynamicWsdl11Definition, 642
EclipseLinkJpaVendorAdapter,
класс, 278
EnvironmentStringPBEConfig, 691
FileSystemResource, 706
FilterChainProxy, 438
FlowHandlerAdapter, 397
FlowHandlerMapping, 396
HessianProxyFactoryBean, 493
HessianServiceExporter, 489
HibernateDaoSupport, класс, 248
HibernateJpaVendorAdapter,
класс, 278
HibernateTemplate, класс, 267
HttpInvokerProxyFactoryBean, 497
HttpInvokerServiceExporter, 495
HttpURLConnectionMessage-
Sender, 650
java.io.File, 365
JavaMailSenderImpl, 701
JaxWsPortProxyFactoryBean, 505
JdbcDaoSupport, класс, 248, 263
JdbcTemplate, класс, 258
JdoDaoSupport, класс, 248
JeeBeanFactoryPostProcessor, 676
JeeEjbBeanFactoryPostProcessor,
676
JmsInvokerProxyFactoryBean, 582
JmsInvokerServiceExporter, 580
JmsServiceExporter, 584
JmsServiceProxy, 584
JmsUtils, 574
JndiObjectFactoryBean, 667
JpaDaoSupport, класс, 248, 280
JpaTemplate, класс, 280
LinkedMultiValueMap, 548
LocalContainerEntityManager-
FactoryBean, класс, 275, 279
LocalEntityManagerFactoryBean,
класс, 275, 279
LocalSessionFactoryBean, класс,
269, 270
LocalStatelessSessionProxy-
FactoryBean, 663
Math, 104
MBeanExporter, 590, 594
MBeanServerConnection, 603
MessageDispatcherServlet, 632, 642
MimeTypeHelper, 705
MultiValueMap, 548
NamedParameterJdbcDaoSupport,
класс, 263
NamedParameterJdbcTemplate,
класс, 258, 262
Naming, 484
ObjectMessage, 574
OpenJpaVendorAdapter, класс, 278
ParameterizedRowMapper,
класс, 261
PersistenceExceptionTranslation-
PostProcessor, класс, 273, 282
ProviderManager, 454
ResponseEntity, 538
RestTemplate, 534
RmiProxyFactoryBean, 485
RmiServiceExporter, 483, 602
SessionFactory, класс, 269
SimpleJaxWsServiceExporter, 500,
503
SimpleJdbcDaoSupport, класс,
248, 263
SimpleJdbcTemplate, класс, 258,
259, 263
SimpleMetadataStrategy, 585
SimpleRemoteStatelessSession-
ProxyFactoryBean, 663
SingleConnectionDataSource,
класс, 252
SoapFaultMappingException-
Resolver, 640
SpringBeanAutowiringSupport, 501

- SqlMapClientDaoSupport, класс, 248
StandardPBEStringEncryptor, 691
TopLinkJpaVendorAdapter, класс, 278
TransactionTemplate, 307
VelocityEngine, 709
VelocityEngineFactoryBean, 709
VelocityEngineUtils, 710
WebServiceGatewaySupport, 657
WebServiceTemplate, 647
класс конфигурации, 193
классы поддержки DAO, 248
поддержки DAO, 247
Клиенты, REST, 532
Коллекции
 <list>, элемент, 92, 93
 <map>, элемент, 93
 <props>, элемент, 93
 <set>, элемент, 92, 93
 внедрение, 92, 97
 свойства-коллекции, 97
Коллекции фиктивных объектов, 59
Компоненты, объявление, 70
Компонент-исполнитель последовательности, 394
Компоненты, 39, 46
 <constructor-arg>, элемент, 75
 автоматическое определение, 188
 автоматическое связывание, 169, 177
 внедрение внутренних компонентов, 89
 внедрение в свойства, 84
 внедрение через конструкторы, 74
 внедрение через методы доступа, 84
 внутренние, 89
 внутри элемента <list>, 94
 возможность ссылаться по идентификаторам, 102
 жизненный цикл, 53
 извлечение из контекста приложения, 53
 инициализация
 и уничтожение, 82, 83
 компоненты-сущности, 273
область действия, 80
объявление, 83
создание фабричными методами, 78
специальные, 140
ссылки на другие компоненты, 87
управляемые сценариями, 156
 внедрение свойств, 162
 пример, 158
экспортирование автономных конечных точек JAX-WS, 502
экспортирование в виде HTTP-служб, 494
экспортирование в виде управляемых компонентов MBean, 589
Компоненты-модели MBean, 589
Компоненты-сущности, 273
Конечные состояния, 398
Конечные точки
 JAX-WS, 500
 абстрактные классы, 623
 автоматическое связывание конечных точек JAX-WS, 500
 использование служб, 647
 маршалеры сообщений, 634, 636
 маршалинг содержимого сообщений, 628, 631
 маршалинг, содержимого сообщений, 628, 631
 настройка, 635
 основанные на модели JDOM, 625
 отображение сообщений, 634
 отправка сообщений, 651
 преобразование исключений в ошибки SOAP, 634, 639
 развертывание службы, 646
 связывание, 632
 создание WSDL-файлов, 641, 644
 шаблоны веб-служб, 648
 маршалеры сообщений, 654
 шлюзы, 656
 экспортирование автономных конечных точек JAX-WS, 502
Конструкторы
 автоматическое связывание, 170, 173



внедрение ссылок, 76
 внедрение через, 74
 Контейнеры, 54, 56, 71
 внедрение зависимостей, 50
 диспетчеры сущностей
 управляемые
 контейнером, 275
 жизненный цикл
 компонент, 53
 Контейнеры обработчиков сообщений, 578
 Контекстные сеансы, 268
 Контекст приложения, 39, 51, 56
 Контроллеры
 @Controller, аннотация, 331
 RESTful, 511
 входной контроллер, 322
 контроллер Hessian, 491
 контроллер главной страницы, 330
 обработка входных данных, 344, 350
 преобразование
 HTTP-сообщений, 528
 создание контроллеров в Spring MVC, 327
 тестирование, 332
 Конфигурационные классы, 193
 Конфигурирование
 Spring в программном коде
 Java, 192
 фреймворка Spring, 71
 Конфликты, разрешение конфликтов в именах управляемых компонентов, 599
 Кэширование
 данных. См. Определение кэширования
 настройка вложенных компонентов для кэширования, 291
 настройка компонентов для кэширования, 289
 Кэширование данных, 282
 декларативное, с помощью аннотаций, 292
 решения, 284
 EHCache, 287
 Кэширование, объектов из JNDI, 697

M

Маршалинг, содержимого сообщений, 628, 631
 Методы
 addAttachment(), 706
 addInline(), 707
 afterPropertiesSet(), 54, 83
 commit(), 303, 304, 306
 convertJmsAccessException(), 574
 create(), 700
 createContainerEntityManagerFactory(), 275
 createEntityManagerFactory(), 275
 createMessage(), 572
 createMimeMessage(), 705
 delete(), 535, 542
 DELETE, 519, 521, 551
 destroy(), 54
 doInTransaction(), 308
 exchange(), 535, 547
 execute(), 308, 535
 get(), 539
 GET, 519, 536
 getAsString(), 136
 getAttribute(), 604
 getBean(), 80
 getContents(), 127
 getFirst(), 539
 getForEntity(), 535, 536, 538
 getForObject(), 535, 536, 537
 getHeaders(), 539
 getInstance(), 79
 getJdbcTemplate(), 264
 getLastModified(), 539
 getObject(), 574
 getStatusCode(), 539
 getWebServiceTemplate(), 657
 hasPermission(), 470
 HEAD, 519
 headForHeaders(), 535
 invoke(), 604
 invokeInternal(), 627
 isCancelled(), 716
 isDone(), 716
 marshalSendAndReceive(), 654
 mergeTemplateToString(), 710
 onApplicationEvent(), 152

OPTIONS, 519
optionsForAllow(), 535
POST, 519, 522
postForEntity(), 535
postForLocation(), 535, 546
postForObject(), 535, 544, 545
postProcessAfterInitialization(), 141
postProcessBeanFactory(), 144
put(), 535, 540
PUT, 519, 520, 540, 551
queryNames(), 603
receive(), 573
registerCustomEditor(), 139
rollback(), 303, 304, 306
send(), 572, 704
setApplicationContext(), 155
setAttribute(), 604
setBeanFactory(), 53, 155
setBeanName(), 53, 154
setFrom(), 704
setText(), 704
setTo(), 704
toUpperCase(), 104
TRACE, 519
асинхронные, 712, 714
безопасность, 463
внедрение методов, 124
 замещение методов, 125
 чтения, 125, 130
возможность ссылаться
по идентификаторам, 102
выполнение по расписанию, 712
доступа, 85
как точки сопряжения, 207
проверка условий безопасности
перед вызовом метода, 467
проверка условий безопасности
после вызова метода, 468
скрытые поля с именем
HTTP-метода в формах, 550
фабричные методы, 78
фильтрация после вызова
метода, 469
экспортирование методов
по их именам, 593
Модель–представление–контрол-
лер, шаблон проектирования, 321
Модули, 55

H

Наследование, 121
свойств, 121
Наследование, в сравнении
с аспектами, 201
Настройка
 EHCache, 287
 Spring Security, 434, 435
 Spring Web Flow, 394
 вложенных компонентов для
 кэширования, 291
 встроенного сервера LDAP, 460
 импортирование внешних
 настроек, 683
 источников данных, 252
 компонентов
 для кэширования, 289
 контроллера Hessian, 491
 механизмов
 Velocity, 370
 механизмы
 FreeMarker, 378
 минимальная настройка
 безопасности, 438
 обработчиков сообщений, 578
 отправки электронной
 почты, 701
 переопределение свойств, 683
 подстановка переменных-
 заполнителей, 683
 реестра последовательности, 395
 служб RMI, 482
 фабрики диспетчера
 сущностей, 274
 элементов Spring Modules, 285
Настройка источников данных, 249
He-Spring компонентов,
внедрение, 132
Неоднозначные зависимости, 180
Неповторимость получаемых
результатов, 312

O

Области видимости
 диалога, 406
 запроса, 406

кадра, 406
последовательности, 406
представления, 406
Область действия компонентов, 80
Облачное хранилище, Amazon S3, 365
Обмен ресурсами, 546
Обмен сообщениями
 ActiveMQ, 563
 POJO, управляемые
 сообщениями, 575
 асинхронный, 556
 брокеры сообщений, 558
 контейнеры обработчиков
 сообщений, 578
 модель «публикация-подписка», 560
 модель «точка-точка», 559
 обработчики сообщений, 578
 отправка сообщений с помощью JmsTemplate, 570
 преимущества Java Message Service, 561
 приемники, 558
 прием сообщений с помощью JmsTemplate, 573
 синхронный, 556
 с помощью Java Message Service, 556
Обработка форм, 350
 входные данные, 353
 выгрузка файлов, 361
 правила проверки, 355
Обработчики сообщений, 578
 настройка, 578
Обратные вызовы, 245
Объектно-реляционное отображение, 57, 266
Объекты доступа к данным, 57, 239
 внедрение в объекты DAO
 Hibernate, 272
 классы поддержки, 247, 248
 классы поддержки JDBC, 263
 на основе JPA, 280
Объявление
 компонентов, 70, 83
 переменных
 в последовательностях, 405

транзакций в XML, 315
Операторы
 ^, 105
 -, 105
 !, 105
 *, 105
 /, 105
 &&, 210
 %, 105
 +, 105
 <, 105
 <=, 105
 ==, 105
 >, 105
 >=, 105
 ||, 210
 and, 105
 eq, 105
 ge, 105
 gt, 105
 le, 105
 lt, 105
 matches, 105
 not, 105
 or, 105
 T(), 104
Определение перечня операций, 618
Определение срезов множества точек сопряжения, 209
Определение формата представления данных, 617
Осведомленность, 153
 BeanNameAware, интерфейс, 154
Ответы
 возврат ресурса в теле ответа, 528
 извлечение метаданных, 538
Открытые компоненты MBean, 589
Отложенная загрузка, 265
Отображение
 сообщений в конечные точки, 634
 объектов в XML, 66
Отображения, 95
Отправка электронной почты, 701

П

Параметры
 contextConfigLocation, 343

- format, 527
HttpMethod, 547
image, 363
Model, 347
ProceedingJoinPoint, 217
в @AspectJ, 228
именованные, 261
передача советам, 218
Пароли, сравнение
с использованием LDAP, 459
Переопределение свойств, 683
Переходы в последовательностях, 398, 402
 глобальные, 404
Поддержка функции «запомнить меня», 462
Подключение к базе данных, 683
Подстановка переменных-заполнителей, 683
Поиск представлений, 334
Полная загрузка, 266
Пользовательские редакторы
свойств, регистрация, 135
Поля форм, связывание
в Velocity, 374
Последовательности, 393
 безопасность, 427
 глобальные переходы, 404
 данные
 в последовательностях, 404
 запросы
 к последовательности, 396
 компонент-исполнитель
 последовательности, 394
 переходы, 398, 402
 реестр последовательности, 395
 сбор данных, 412
 состояния, 397
Последующая операция (after advice), 46
Постобработка, 141
Правила отмены транзакций, 315
Правила распространения, 310
Практике, 246
Предварительная операция (before advice), 46
Представление, 511
 ресурсов REST, 523
- Представления
 FreeMarker, механизм
 шаблонов, 377
Velocity, механизм шаблонов, 369
 безопасность, 447
 внутренние, 336
 отображение, 348
 поиск, 334
Преобразование исключений
в ошибки SOAP, 634, 639
Преобразование
HTTP-сообщений, 528
Привилегии, 449
Прием извещений, 609
Приемник, настройка
по умолчанию, 572
Приемники, 558
 очереди, 559
 темы, 559
Примеры
 Audience, 212
 Auditorium, 82
 Contestant, 222
 Instrumentalist, 84, 171, 189
 JDBC, 47
 juggler, 73
 Juggler, 194
 KnifeJuggler, 185
 knight, 39, 43
 Magician, 229
 MindReader, 218
 OneManBand, 93, 97
 PoeticJuggler, 76, 78, 174
 Spitter, 259, 306
 Spring Idol, 70
 Stage, 79
 ticket, 80
 заказ пиццы, 407
Проверка
 входных данных, 355
 правила, 355
Проверка условий безопасности
перед вызовом метода, 467
Проверка условий безопасности
после вызова метода, 468
Программное управление
транзакциями, 306



Производительность
воспринимаемая, 715
фактическая, 714

Проксирование
сессионных компонентов (EJB 2.x), 663
управляемых компонентов MBean, 605

Пространства имен
aop, 45, 226, 316
beans, 71, 72
context, 65, 72, 177, 597, 684
jee, 72, 249, 279, 666, 695
jms, 65, 72, 578
lang, 72, 160
mvc, 72, 326
oxm, 72
p, 91
security, 436
tx, 72, 309, 316
util, 72
конфигурационное Spring Security, 434, 435

P

Регистрация пользовательских редакторов свойств, 135

Редакторы свойств, пользовательские, регистрация, 135

Реестр последовательности, 395

Ресурсы
REST, 510, 511
возврат ресурса в теле ответа, 528
изменение, 540
клиенты REST, 532
контроллеры RESTful, 511
обмен, 546
представление, 523
прием объектов в ответах, 543
прием ресурса в теле ответа, 531
создание, 543
удаление, 542
чтение ресурсов, 536, 537

Родительские компоненты, 118

C

Сборщики информации MBean, 593

InterfaceBasedMBeanInfo-Assembler, 596

MetadataMBeanInfoAssembler, 597

MethodExclusionMBeanInfo-Assembler, 594

MethodNameBasedMBeanInfo-Assembler, 593

Свойства
algorithm, 691
annotatedClasses, 271
connectionFactory, 570, 584
database, 279
dataSource, 264, 269, 303, 696
defaultFault, 641
defaultView, 528
destination, 584
em, 281
exceptionMappings, 641
favorParameter, 527
favorPathExtension, 527
hibernateProperties, 269
host, 702
ignoreAcceptHeader, 527
jdbcTemplate, 264
jdbc.url, 685
jndiName, 664
location, 690
locationUri, 643
mappingResources, 269
mediaTypes, 526
metadataStrategy, 585
objectName, 606
packagesToScan, 270
password, 691, 702
persistenceUnitName, 277
port, 702
portTypeName, 643
proxyInterface, 606
registrationBehaviorName, 600
registryHost, 484
registryPort, 484
schema, 642
server, 592, 606
service, 584
serviceInterface, 486, 497, 583, 584

- serviceUrl, 486, 498, 602
sessionFactory, 272, 303
transactionManagerName, 306
username, 702
velocityProperties, 709
viewResolvers, 527
wsdlDocumentUrl, 506
автоматическое связывание, 169
внедрение в, 84
внедрение пустых значений, 99
внешние файлы
с настройками, 145
возможность ссылаться
по идентификаторам, 102
наследование, 121
общие абстрактные, 122, 123
отсутствующие, 685
переопределение наследуемых,
122, 123
переопределение свойств, 683
подстановка переменных-
заполнителей, 683
свойства-коллекции, 97
системные, 686
Свойство, иные значения
термина, 98
Связывание
Enterprise JavaBeans, 661
 внедрение в компоненты
 Spring, 669
 проксирование сеансовых
 компонентов (EJB 2.x), 663
@Inject, аннотация, 184
автоматическое связывание, 169
неоднозначные зависимости, 180
объектов из JNDI, 692
смешивание автоматического
и явного связывания, 175
с помощью аннотаций, 177
Связывание компонентов, 118
 внедрение методов, 124
 объявление дочерних
 компонентов, 118
 абstractные компоненты, 119
 объявление родительских
 компонентов, 118
 абstractные компоненты, 119
Сеансовые компоненты
 EJB 2.x, 663, 669
 EJB 3, 667
Сервер MBean, 590
Сервлет-фильтры, 437
Сервлеты
 DispatcherServlet, 322
 сервлет-фильтры, 437
Синхронные
взаимодействия, 556, 561
Сквозные задачи, 41, 199
Слабая связанность, 32, 88
Слои доступа к данным, 239
 скрытие за интерфейсами, 240
Слои доступа к данным, 243
Службы
 HTTP Invoker, 494
 доступ к службам Hessian
 и Burlap, 492
 доступ к службам по протоколу
 HTTP, 497
 конечные точки JAX-WS, 500
 настройка служб RMI, 482
 удаленные, 477
 экспортирование служб
 Burlap, 492
 экспортирование служб
 Hessian, 489
События
 в приложении, 150
 прием, 152
 публикация, 151
Советы, 202
 выполняемые до, 202
 выполняемые и до, и после
 вызыва, 202
 выполняемые после, 202
 выполняемые после
 исключения, 202
 выполняемые после успешного
 вызыва, 202
 написаны на Java, 206
 передача параметров, 218
Сообщения
 интернационализация, 147
 маршалеры сообщений, 634, 636
 маршалинг содержимого, 628,
 631

обработка в веб-службах, 623
 отображение в конечные
 точки, 634
 отправка, 651
 преобразование исключений
 в ошибки SOAP, 634, 639
 примеры XML-сообщений, 616
Состояние, 511
Состояния-действия, 398
Состояния-
 подпоследовательности, 398
Состояния
 последовательностей, 397, 398
 действия, 398
 конечные, 398
 подпоследовательности, 398
 представления, 398
 решения, 398
Состояния-представления, 398
Состояния-решения, 398
Специальные компоненты, 140
 осведомленность, 153
Срезы множества точек
 сопряжения, 203
 определение, 209
Ссылки на компоненты, 87
Ссылки на объекты, внедрение
 через конструкторы, 76
Стандартные компоненты
 MBean, 588
Строковый шифратор, 690
Сценариями управляемые
 компоненты, 156
 внедрение свойств, 162
 пример, 158

T

Тесная связь, 693
Тестирование
 контроллеров, 332
 модуль поддержки
 тестирования
Типы и язык выражений Spring, 104
Точка внедрения, 46
Точки внедрения
 @annotation(), 208
 @args(), 208
 args(), 208

execution(), 208
 @target(), 208
 target(), 208
 this(), 208
 @within(), 208
 within(), 208
Точки сопряжения, 202
 методы, 207
Транзакции, 296
 ACID, 299
 Java Persistence API, 300
 Java Transaction API, 300
 @Transactional, аннотация, 309
 атомарность, 299
 атрибуты транзакций, 309
 время ожидания, 315
 декларативное управление
 транзакциями, 309
 диспетчеры, 301
 долговечность, 299
 знакомство, 297
 изолированность, 299
 неповторимость получаемых
 результатов, 312
 непротиворечивость, 299
 объявление в XML, 315
 определение с помощью
 аннотаций, 318
 поддержка в Spring, 300
 правила отмены, 315
 правила распространения, 310
 программное управление
 транзакциями, 306
 только для чтения, 314
 управляемые контейнером, 300
 уровни изоляции, 312
 чтение неподтвержденных
 данных, 312
 чтение фантомных данных, 312

У

Удаленные службы, 477
 Burlap, 478
 Hessian, 478
 Hessian и Burlap, 488
 HTTP Invoker, 478, 494
 вызов удаленных методов, 478

- доступ к службам по протоколу HTTP, 497
конечные точки JAX-WS, 500
обзор, 477
Уничтожение компонентов, 82, 83
Управляемые атрибуты, 590
Управляемые компоненты
 определение экспортруемых операций и атрибутов с помощью интерфейсов, 596
 агент MBean, 590
 динамические компоненты MBean, 588
 доступ к удаленным компонентам MBean, 603
 извещения, 606
 компоненты-модели MBean, 589
 объявление с помощью аннотаций, 597
 открытые компоненты MBean, 589
 проксирование, 605
 разрешение конфликтов в именах компонентов, 599
 сборщики информации MBean, 593
 сервер MBean, 590
 стандартные компоненты MBean, 588
 удаленные, 602
 экспортование методов по их именам, 593
 экспортование с помощью Spring, 589
 экспортование удаленных компонентов MBean, 602
Управляемые сценариями компоненты, 156
 внедрение свойств, 162
 пример, 158
Уровни изоляции, 312
Установка фреймворка Spring Web Flow, 393
- Ф**
Фабрика компонентов, 56
- Фабрики компонентов, 51
Фабрики сеансов, 268
Фабричные методы, создание компонентов, 78
Файловая система, выгрузка файлов, 364
Файлы
 выгрузка, 361
 сохранение в Amazon S3, 365
 сохранение в файловой системе, 364
Фактическая производительность, 714
Фильтрация
 <context:component-scan>, элемент, 191
 <context:exclude-filter>, элемент, 191
 <context:include-filter>, элемент, 191
Фильтрация результатов после вызова метода, 469
Фоновые задания, 711
Форматирование в Velocity, 372
Формы
 RESTful, 550
 аутентификации, 439
 выгрузка файлов, 361
 добавление поля выгрузки файла, 361
 обработка, 350
 обработка данных, 353
 определение представлений, 351
 правила проверки, 355
 представления форм, 351
 проверка входных данных, 355
 регистрации, 350
 скрытое поле с именем HTTP-метода, 550
 тип содержимого, 361
Фреймворки, аспектно-ориентированного программирования, 205
- Х**
Хранение данных, 239
 источники данных, 252
 кэширование решения, 284



настройка источников данных, 249
обзор, 239
обратные вызовы, 245
с помощью Hibernate, 267
шаблоны, 244, 245, 246

Ч

Чтение неподтвержденных данных, 312
Чтение фантомных данных, 312

Ш

Шаблонный код, 256
устранение, 47, 50
Шаблонный метод (Template Method), шаблон проектирования, 244
Шаблоны, 47, 50, 245
 CciTemplate, класс, 246
 HibernateTemplate, класс, 246
 JdbcTemplate, 49
 JdbcTemplate, класс, 246
 JdoTemplate, класс, 246
 JpaTemplate, класс, 246
 NamedParameterJdbcTemplate, класс, 246
 SimpleJdbcTemplate, 49
 SimpleJdbcTemplate, класс, 246
 SqlMapClientTemplate, класс, 246
веб-служб, 648
 шилзы, 656
электронная почта, 708
Шаблоны доступа к данным, 244
Шаблоны проектирования,
Шаблонный метод (Template Method), 244
Шифрование, 459, 690

Э

Электронная почта
 вложения, 705
 отправка, 701
 содержимое в формате HTML, 707

создание писем, 704
шаблоны, 708
Электронные таблицы Excel, 384

Я

Язык выражений (Spring Expression Language, SpEL), 100
T(), оператор, 104
выборка элементов коллекций, 114
доступ к элементам коллекций, 112
и типы, 104
литералы, 100
логические операции, 108
математические операции, 106
обработка коллекций, 111
операции, 105
отображение коллекций, 114
регулярные выражения, 110
сравнение значений, 107
ссылки по идентификаторам, 102
условные вычисления, 109

А

AbstractDom4jPayloadEndpoint, класс, 624
AbstractDomPayloadEndpoint, класс, 624
AbstractExcelView, класс, 384
AbstractJDomPayloadEndpoint, класс, 624
AbstractJmsMessageDrivenBean, класс, 670
AbstractMarshallingPayloadEndpoint, класс, 624
AbstractMessageDrivenBean, класс, 670
AbstractSaxPayloadEndpoint, класс, 624
AbstractStatefulSessionBean, класс, 670
AbstractStatelessSessionBean, класс, 670
AbstractStaxEventPayloadEndpoint, класс, 624

- AbstractStaxStreamPayloadEndpoint, класс, 624
AbstractXomPayloadEndpoint, класс, 624
abstract, атрибут, 118
Accept, заголовок, 525, 548
access, атрибут, 445, 450
ACID, определение, 299
ActiveMQ, 563
addAttachment(), метод, 706
addInline(), метод, 707
Adobe AIR, 63
Adobe Flex, 63
ADO.NET, 63
afterPropertiesSet(), метод, 54, 83
algorithm, свойство, 691
Amazon S3, 365
and, оператор, 105
annotatedClasses, свойство, 271
AnnotationAwareAspectJAutoProxy-Creator, класс, 226
AnnotationSessionFactoryBean, класс, 270
аор, пространство имен, 45, 226, 316
Apache Struts, 58
Apache Tiles, 338
 <definition>, элемент, 351
 определение шаблона, 339
ApplicationContextAwareProcessor, интерфейс, 143
ApplicationContextAware, интерфейс, 53, 153
ApplicationContext, интерфейс, 51, 207
 publishEvent(), метод, 151
arg-names, атрибут, 220
args(), указатель для точки внедрения, 208
AspectJ, 192, 205, 206
 @AspectJ, 224
 внедрение аспектов, 231
 в сравнении с Spring AOP, 207
AspectJ, аспекты, 135
AspectJ, язык выражений, 46, 65
ASP.NET, 63
AsyncResult, класс, 715
AtomFeedHttpMessageConverter, 530
Audience, пример, 212
Auditorium, пример, 82
authentication-failure-url, атрибут, 440
authorities-by-username-query, атрибут, 457
authorities, атрибут, 453
auto-config, атрибут, 439
- ## B
- base-package, атрибут, 189
base-path, атрибут, 395
BasicDataSource, класс, 250
 driverClassName, свойство, 251
 initialSize, свойство, 251
 maxActive, свойство, 251
 maxIdle, свойство, 251
 maxOpenPreparedStatements, свойство, 251
 maxWait, свойство, 251
 minEvictableIdleTimeMillis, свойство, 251
 minIdle, свойство, 251
 password, свойство, 251
 poolPreparedStatements, свойство, 251
 url, свойство, 251
 username, свойство, 251
BeanCounter, 144
BeanFactoryAware, интерфейс, 53, 153
BeanFactory, интерфейс, 51
BeanNameAware, интерфейс, 53, 154
BeanNameUrlHandlerMapping, 328
BeanNameViewResolver, 334
BeanPostProcessor, интерфейс, 54, 141, 144
beans, пространство имен, 71, 72
BufferedImageHttpMessage-Converter, 530
buildExcelDocument(), метод, 386
buildPdfDocument(), метод, 389
Burlap, 58, 478, 488
 доступ к службам, 492
 экспортирование служб, 492
 экспортирование функциональности компонентов, 489
Burlap в сравнении с Hessian, 488



BurlapServiceExporter, класс, 492
business-interface, атрибут, 700
ByteArrayHttpMessageConverter, 530

C

cache, атрибут, 697
catch, блоки, 241, 244
CciDaoSupport, класс, 248
CciLocalTransactionManager, 302
CciTemplate, класс, 246
Central Authentication Service и Spring Security, 434
ClassPathXmlApplicationContext, 51
ClassPathXmlApplicationContext, класс, 39
class, атрибут, 74
commit(), метод, 303, 304, 306
CommonsHttpMessageSender, класс, 650
CommonsMultipartResolver, класс, 367
connectionFactory, атрибут, 579
connectionFactory, свойство, 570, 584
ConnectorServerFactoryBean, класс, 602
ContentNegotiatingViewResolver, 334, 525
 определение формата
 представления данных, 526
 поиск представлений, 527
content, атрибут, 352
Contestant, пример, 222
contextConfigLocation, параметр, 343
ContextLoaderListener, интерфейс, 343
context, пространство имен, 65, 72, 177, 597, 684
 <context:property-placeholder>, элемент, 684
«contract-first», модель разработки веб-служб, 613, 614, 616
 использование служб, 647
 конечные точки, основанные на модели JDOM, 625
 маршалеры сообщений, 634, 636
 маршалинг содержимого сообщений, 628, 631

настройка, 635
отображение сообщений в конечные точки, 634
отправка сообщений, 651
преобразование исключений в ошибки SOAP, 634, 639
развертывание службы, 646
связывание служб, 632
создание WSDL-файлов, 641, 644
создание примеров XML-сообщений, 616
шаблоны веб-служб, 648
 маршалеры сообщений, 654
 шлюзы, 656
«contract-last», модель разработки веб-служб, 613
ControllerBeanNameHandler-Mapping, 328
ControllerClassNameHandler-Mapping, 329
convertJmsAccessException(), метод, 574
CORBA, 481
CosMailSenderImpl, класс, 701
CouchDB, 64
createContainerEntityManager-Factory(), метод, 275
createEntityManagerFactory(), метод, 275
createMessage(), метод, 572
createMimeMessage(), метод, 705
create(), метод, 700
CronTriggerBean, 154
cron, атрибут, 713
CRUD, операции, 520
CSS (Cascading Style Sheets – каскадные таблицы стилей), 388
CustomEditorConfigurer, 139

D

DataAccessException, исключение, 243
database, свойство, 279
DataSourceTransactionManager, 302, 303
data-source, атрибут, 455
DataSource, извлечение из JNDI, 693

DataSource, интерфейс, 259, 303
 dataSource, свойство, 264, 269, 303, 696
 db4o, 64
 DefaultAnnotationHandler-Mapping, 329
 default-autowire, атрибут, 175
 default-destroy-method, атрибут, 83
 defaultFault, свойство, 641
 default-impl, атрибут, 223
 defaultImpl, атрибут, 230
 default-init-method, атрибут, 83
 DefaultMethodSecurityExpression-Handler, класс, 471
 default-ref, атрибут, 698
 defaultView, свойство, 528
 delegate-ref, атрибут, 223
 DelegatingFilterProxy, класс, 437
 DELETE, HTTP-метод, 519, 521, 551
 HiddenHttpMethodFilter, 552
 delete(), метод, 535, 542
 destination, свойство, 584
 destroy-method, атрибут, 82
 destroy(), метод, 54
 DispatcherServlet, 322, 334
 загрузка контекста приложения, 343
 и Spring Web Flow, 394
 настройка, 324
 шаблоны URL, 325
 DisposableBean, интерфейс, 54, 83
 doInTransaction(), метод, 308
 DriverManagerDataSource, класс, 252
 DynamicWsdl11Definition, класс, 642

E

EclipseLinkJpaVendorAdapter, класс, 278
 EHCache, 285
 EJB 2.x
 Enterprise JavaBeans
 с поддержкой Spring, 670
 в сравнении с EJB 2.x, 661
 сессионные компоненты, проксируирование, 669
 сложности, 673
 EJB 3, 673

Pitchfork, 674
 внедрение ресурсов
 с помощью аннотаций, 676
 объявление перехватчиков
 с помощью аннотаций, 678
 в сравнении с EJB 3, 661
 сессионные компоненты, объявление, 667
 else, атрибут, 401
 em, свойство, 281
 Enterprise JavaBeans
 и транзакции, 300
 Enterprise JavaBeans (EJB), 660
 сложности, 673
 Enterprise JavaBeans (EJB 2.x)
 с поддержкой Spring, 670
 EntityManagerFactory, интерфейс, 274, 277, 279, 305
 EntityManager, интерфейс, 275
 EnvironmentStringPBEConfig, класс, 691
 eq, оператор, 105
 exceptionMappings, свойство, 641
 exchange(), метод, 535, 547
 execute(), метод, 308, 535
 execution(), указатель для точки внедрения, 208
 exposeRequestAttributes, свойство, 373, 380
 exposeSessionAttributes, свойство, 373, 380
 expression, атрибут, 191, 400

F

FactoryBean, интерфейс, 269
 factory-method, атрибут, 79, 80, 234
 failOnExisting, 601
 favorParameter, свойство, 527
 favorPathExtension, свойство, 527
 FileSystemResource, класс, 706
 FileSystemXmlApplicationContext, 51
 FilterChainProxy, класс, 438
 Filter, интерфейс, 437
 fixedDelay, атрибут, 713
 fixedRate, атрибут, 713
 FlowHandlerAdapter, класс, 397
 FlowHandlerMapping, класс, 396



format, параметр, 527
 FormHttpMessageConverter, 530, 542
 FreeMarkerConfigurer,
 компонент, 378
 freemarkerSettings, свойство, 379
 FreeMarkerViewResolver, 335
 FreeMarkerViewResolver,
 компонент, 379
 FreeMarker, механизм шаблонов, 377
 настройка, 378
 разрешение представлений, 379
 связывание полей форм, 380
 создание представлений, 377
 Future, интерфейс, 715

G

Gemini Blueprint, 62
 getAsText(), метод, 136
 getAttribute(), метод, 604
 getBean(), метод, 80
 getContents(), метод, 127
 getFirst(), метод, 539
 getForEntity(), метод, 535, 536, 538
 getForObject(), метод, 535, 536, 537
 getHeaders(), метод, 539
 GET, HTTP-метод, 519, 536
 getInstance(), метод, 79
 getJdbcTemplate(), метод, 264
 getLastModified(), метод, 539
 getObject(), метод, 574
 getStatusCode(), метод, 539
 getWebServiceTemplate(), метод, 657
 get(), метод, 539
 ge, оператор, 105
 GigaSpaces, 285
 Grails, фреймворк, 64
 group-search-base, атрибут, 458
 group-search-filter, атрибут, 458
 gt, оператор, 105

H

hasAnyRole(), выражение SpEL, 445
 hash, атрибут, 459
 hasIpAddress(), выражение SpEL, 445
 hasPermission(), метод, 470
 hasRole(), выражение SpEL, 445

headers, атрибут, 529
 headForHeaders(), метод, 535
 HEAD, HTTP-метод, 519
 Hessian, 58, 478, 488
 в сравнении с Burlap, 488
 доступ к службам, 492
 настройка контроллера, 491
 экспортирование служб, 489
 экспортирование функциональности компонентов, 489
 HessianProxyFactoryBean, класс, 493
 HessianServiceExporter, класс, 489
 Hibernate, 57, 132, 267
 @Entity, аннотация, 270
 HibernateTemplate, класс, 246
 доступ к данным без использования классов шаблонов, 271
 иерархия исключений, 242
 интеграция с фреймворком Spring, 265
 и транзакции, 302, 303
 каскадирование, 266
 контекстные сеансы, 268
 обзор, 267
 объявление фабрики сеансов, 268
 отложенная загрузка, 265
 полная загрузка, 266
 HibernateDaoSupport, класс, 248
 HibernateJpaVendorAdapter, класс, 278
 hibernateProperties, свойство, 269
 HibernateTemplate, класс, 246, 267
 HibernateTransactionManager, 302
 HiddenHttpMethodFilter, 552
 host, свойство, 702
 HTTP Invoker, 478
 доступ к службам по протоколу HTTP, 497
 экспортирование компонентов в виде HTTP-служб, 494
 HttpInvokerProxyFactoryBean, класс, 497
 HttpInvokerServiceExporter, класс, 495
 HttpMethod, параметр, 547
 HttpServletRequest, 436
 HttpURLConnectionMessageSender, класс, 650

I

iBATIS, 132
и транзакции, 301
iBATIS SQL Maps, 58
id, атрибут, 73, 91
ignoreAcceptHeader, свойство, 527
ignoreExisting, 601
ignore-resource-not-found, атрибут, 686
ignore-unresolvable, атрибут, 686
IllegalArgumentException, исключение, 209
ImageUploadException, исключение, 365
image, параметр, 363
implement-interface, атрибут, 223
InitializingBean, интерфейс, 54, 83
initialSize, свойство, 251
init-method, атрибут, 54, 82
InstanceAlreadyExistsException, исключение, 600
Instrumentalist, пример, 84, 171, 189
InterfaceBasedMBeanInfo-Assembler, 596
InternalResourceViewResolver, 335, 336
invokeInternal(), метод, 627
invoke(), метод, 604
IOException, исключение, 534
isAnonymous(), выражение SpEL, 445
isAuthenticated(), выражение SpEL, 445
isCancelled(), метод, 716
isDone(), метод, 716
isFullyAuthenticated(), выражение SpEL, 445
ISOLATION_DEFAULT, уровень изоляции, 313
ISOLATION_READ_COMMITTED, уровень изоляции, 313
ISOLATION_READ_UNCOMMITTED, уровень изоляции, 313
ISOLATION_REPEATABLE_READ, уровень изоляции, 313

ISOLATION_SERIALIZABLE, уровень изоляции, 313
isRememberMe(), выражение SpEL, 445
iText PDF, библиотека, 390

J

JAR-файлы, 55
JasperReportsViewResolver, 335
Jasypt, 683, 689
Java
jee, пространство имен, 72
jms, пространство имен, 72
Math, класс, 104
внедрение зависимостей в конфигурации на Java, 195
конфигурирование Spring, 192
упрощение разработки, 50
Java API, шаблонный код, 47
JavaBeans
внедрение зависимостей, 35
жизненный цикл, 53
и автоматическое определение, 188
контейнер, 56
контейнеры, 50
область действия, 80
объявление, 70, 83
java.beans.PropertyEditor, интерфейс, 136
Java Community Process (JSR-330), организация, 183
Java Data Objects, 57, 274
java.io.File, класс, 365
JavaMailSenderImpl, класс, 701
Java Management Extensions, 588
извещения, 606
и удаленные взаимодействия, 602
Java Message Service, 58, 556
JmsTemplate, 566
архитектура, 557
борьба с разбуханием кода, 566
брокеры сообщений, 558
введение, 556
очереди, 559
преимущества, 561
приемники, 558



- темы, 559
- Java Messaging Service
 - JMS Invoker, 580
 - POJO, управляемые сообщениями, 575
- Java Naming and Directory Interface, 692
 - кэширование объектов из JNDI, 697
- Java Persistence API, 57, 274, 300
 - и транзакции, 304
- Java Transaction API, 300, 305
 - и транзакции, 305
- java.util.Collection, интерфейс, 92, 95
- java.util.Map, интерфейс, 93, 96, 263
- java.util.Properties, интерфейс, 93, 98
- java.util.Set, интерфейс, 95
- javax.inject, пакет, 185
- Jaxb2RootElementHttpMessageCon verter, 530
- JAX-WS, 58
 - автоматическое связывание конечных точек, 500
 - конечные точки, 500
 - экспортирование автономных конечных точек, 502
- JaxWsPortProxyFactoryBean, класс, 505
- JBoss, 249
- JBoss AOP, 205
- JBoss Cache, 285
- JConsole, и JMX, 590
- JCS, 285
- JDBC, 252
 - JdbcTemplate, класс, 246
 - NamedParameterJdbcTemplate, класс, 246
 - SimpleJdbcTemplate, класс, 246
 - SQLException, исключение, 241
 - доступ к данным, 265
 - иерархия исключений JDBC в сравнении с иерархией исключений Spring, 242
 - извлечение строк в базе данных, 256
 - источники данных, 252
 - и транзакции, 301
 - обновление строк в базе
- данных, 255
- разбухание кода, 254
- шаблонный код, 47, 256
- шаблоны, 258
- JdbcDaoSupport, класс, 248, 263
- JdbcTemplate, класс, 49, 246, 258
- jdbcTemplate, свойство, 264
- jdbc.url, свойство, 685
- JdoDaoSupport, класс, 248
- JdoTemplate, класс, 246
- JdoTransactionManager, 302
- JeeBeanFactoryPostProcessor, класс, 676
- JeeEjbBeanFactoryPostProcessor, класс, 676
- jee, пространство имен, 72, 249, 279, 666, 695
- JetS3t, библиотека, 365
- JmsException, исключение, 569
- JMSEException, исключение, 568
- JMS Invoker, 580
- JmsInvokerProxyFactoryBean, класс, 582
- JmsInvokerServiceExporter, класс, 580
- JmsServiceExporter, класс, 584
- JmsServiceProxy, класс, 584
- JmsTemplate, 566
 - receive(), метод, 573
 - send(), метод, 572
 - отправка сообщений, 570
 - прием сообщений, 573
- JmsTransactionManager, 302
- JmsTransactionManager102, 302
- JmsUtils, класс, 574
- jms, пространство имен, 65, 72, 578
- JMX Messaging Protocol, 601
- JNDI, 692
 - <jee:jndi-lookup>, элемент, 249
 - источники данных, 249
- jndi-name, атрибут, 250, 695, 700
- jndiName, свойство, 664
- JndiObjectFactoryBean, класс, 667
- JpaDaoSupport, класс, 248, 280
- JpaDialect, интерфейс, 305
- JpaTemplate, класс, 246, 280
- JpaTransactionManager, 302, 304
- JSF, 58

JSON, 66
JSP, теги, 132
JSR-160, спецификация, 601
jsr250-annotations, атрибут, 465
JtaTransactionManager, 302, 305
juggler, пример, 73
Juggler, пример, 194

K

Kerberos, 64
key-ref, атрибут, 97
key, атрибут, 97, 591
KnifeJuggler, пример, 185

L

lang, пространство имен, 72, 160
lazy-init, атрибут, 665, 699
ldif, атрибут, 461
le, оператор, 105
Lightweight Directory Access Protocol и Spring Security, 434
Lingo, библиотека, 583
LinkedMultiValueMap, класс, 548
LocalContainerEntityManagerFactoryBean, класс, 275, 279
LocalEntityManagerFactoryBean, класс, 275, 279
LocalSessionFactoryBean, класс, 269, 270
LocalStatelessSessionProxyFactoryBean, класс, 663
locationUri, свойство, 643
location, свойство, 146, 690
login-processing-url, атрибут, 440
login, атрибут, 440
logout-url, атрибут, 443
lookup-on-startup, атрибут, 698
lt, оператор, 105

M

Magician, пример, 229
MailSender, интерфейс, 701, 703
MailSession, интерфейс, 703
MalformedURLExceptions, исключение, 481

MappingJacksonHttpMessageConverter, 530, 542
mappingResources, свойство, 269
mapping, атрибут, 326
Marshaller, интерфейс, 636
MarshallingHttpMessageConverter, 530
marshalSendAndReceive(), метод, 654
matches, оператор, 105
Math, класс, 104
maxActive, свойство, 251
maxIdle, свойство, 251
maxOpenPreparedStatements, свойство, 251
maxWait, свойство, 251
MBeanExporter, класс, 590, 594
registrationBehaviorName, свойство, 600
server, свойство, 592
MBeanServerConnectionFactoryBean, 603
MBeanServerConnection, класс, 603
mediaTypes, свойство, 526
mergeTemplateToString(), метод, 710
MessageCreator, интерфейс, 572
MessageDispatcherServlet, класс, 632, 642
MessageDrivenBean, интерфейс, 576
MessageListener, интерфейс, 576
MessageSource, интерфейс, 148
MetadataMBeanInfoAssembler, 597
metadataStrategy, свойство, 585
MethodExclusionMBeanInfoAssembler, 594
MethodNameBasedMBeanInfoAssembler, 593
methodName, атрибут, 290
method, атрибут, 521
MimeMessageHelper, класс, 705
MindReader, пример, 218
minEvictableIdleTimeMillis, свойство, 251
minIdle, свойство, 251
Mockito, 333
Model-View-Controller, 58



model, атрибут, 399
Model, параметр, 347
MSMQ, 63
MultipartResolver, интерфейс, 367
Multipurpose Internet Mail
Extensions, 705
MultiValueMap, класс, 548
mvc, пространство имен, 72, 326

N

NamedParameterJdbcDaoSupport,
класс, 248, 263
NamedParameterJdbcTemplate,
класс, 246, 258, 262
name, атрибут, 316, 416, 453
NamingException, исключение, 693
Naming, класс, 484
new, ключевое слово, 53
NHibernate, 63
NoSuchBeanDefinitionException,
исключение, 180
NotificationListener, интерфейс, 609
NotificationPublisherAware,
интерфейс, 607
NotModifiedException,
исключение, 540
not, оператор, 105
NullPointerException, исключение,
104, 176

O

ObjectMessage, класс, 574
objectName, атрибут, 598
objectName, свойство, 606
Object Request Broker, 481
OC4JtaTransactionManager, 302
onApplicationEvent(), метод, 152
OneManBand, пример, 93, 97
on-exception, атрибут, 404
on, атрибут, 403
OpenID и Spring Security, 434
OpenJpaVendorAdapter, класс, 278
OpenSymphony OSCache, 285
optionsForAllow(), метод, 535
OPTIONS, HTTP-метод, 519
org.springframework.beans.

propertyeditors.ClassEditor, 136
org.springframework.beans.property-
editors.CustomDateEditor, 136
org.springframework.beans.
propertyeditors.FileEditor, 137
org.springframework.beans.
propertyeditors.LocaleEditor, 137
org.springframework.
beans.propertyeditors.
StringArrayPropertyEditor, 137
org.springframework.beans.
propertyeditors.StringTrimmer-
Editor, 137
org.springframework.beans.
propertyeditors.
URLEditor, 137
org.springframework.context.
ApplicationEvent, абстрактный
класс, 152
org.springframework.context.
ApplicationListener, интерфейс, 152
or, оператор, 105
OSGi Blueprint Container, 62
OXM, 66
oxm, пространство имен, 72

P

packagesToScan, свойство, 270
ParameterizedRowMapper,
класс, 261
params, атрибут, 351
parent, атрибут, 118
password-attribute, атрибут, 459
password, атрибут, 453
password, свойство, 691, 702
path-type, атрибут, 443
path, атрибут, 353, 396
pattern, атрибут, 438, 443
PDF (Portable Document
Format – переносимый формат
документов), 383
PersistenceExceptionTranslation-
PostProcessor, класс, 273, 282
PersistenceProvider, интерфейс, 275,
277
persistenceUnitName, свойство, 277
persistence.xml, файл, 276

physicalName, атрибут, 566
Pitchfork, 674
внедрение ресурсов с помощью аннотаций, 676
объявление перехватчиков с помощью аннотаций, 678
поддерживаемые аннотации, 675
PoeticJuggler, пример, 76, 78, 174
pointcut-ref, атрибут, 216
pointcut, атрибут, 215
poolPreparedStatements, свойство, 251
portTypeName, свойство, 643
port, свойство, 702
postForEntity(), метод, 535
postForLocation(), метод, 535, 546
postForObject(), метод, 535, 544, 545
POST, HTTP-метод, 519, 522
 HiddenHttpMethodFilter, 552
postProcessAfterInitialization(), метод, 141
postProcessBeanFactory(), метод, 144
pre-post-annotations, атрибут, 466
primary, атрибут, 172
private, ключевое слово и аннотация @Autowired, 179
ProceedingJoinPoint, параметр, 217
PROPAGATION_MANDATORY, правило поведения, 311
PROPAGATION_NESTED, правило поведения, 311
PROPAGATION_NEVER, правило поведения, 311
PROPAGATION_NOT_SUPPORTED, правило поведения, 311
PROPAGATION_REQUIRED, правило поведения, 311
PROPAGATIONQUIRES_NEW, правило поведения, 311
PROPAGATION_SUPPORTS, правило поведения, 312
properties-ref, атрибут, 686
PropertyPlaceholderConfigurer, 145
property, атрибут, 448
ProviderManager, класс, 454
Provider, интерфейс, 185

proxy-interface, атрибут, 697
proxyInterface, свойство, 606
PUT, HTTP-метод, 519, 520, 540, 551
 HiddenHttpMethodFilter, 552
put(), метод, 535, 540
р, пространство имен, 91

Q

queryNames(), метод, 603

R

receive(), метод, 573
refresh-check-delay, атрибут, 164
ref, атрибут, 78, 102, 214
registerCustomEditor(), метод, 139
registrationBehaviorName, свойство, 600
REGISTRATION_FAIL_ON_EXISTING, 600
REGISTRATION_IGNORE_EXISTING, 600
REGISTRATION_REPLACE_EXISTING, 600
registration, атрибут, 600
registryHost, свойство, 484
registryPort, свойство, 484
RemoteAccessException, исключение, 480
RemoteException, исключение, 479, 481
Remote Method Invocation, 58
replaceExisting, 601
required, атрибут, 180
requires-channel, атрибут, 446
ResourceBundleMessageSource, 149
ResourceBundleViewResolver, 335
ResourceHttpMessageConverter, 530
resource-ref, атрибут, 250, 695
ResponseEntity, класс, 538
REST, 66, 510
 URL в стиле RESTful, 514
 введение, 510
 возврат ресурса в теле ответа, 528
 выполнение операций, 519
 извлечение метаданных

из ответов, 538
изменение ресурсов, 540
клиенты, 532
контроллеры, 511
параметризованные URL, 517
поддержка в Spring, 511
представление ресурсов, 523
прием ресурса в теле ответа, 531
создание новых ресурсов, 543
удаление ресурсов, 542
формы, 550
чтение ресурсов, 536, 537
RestClientException,
исключение, 538
RestTemplate, класс, 534
RMI, 58, 478, 481
RmiProxyFactoryBean, класс, 485
RmiRegistryFactoryBean, 602
RmiServiceExporter, класс, 483, 602
rollback(), метод, 303, 304, 306
root, атрибут, 460
RSS, 66
RssChannelHttpMessage-
Converter, 531
Ruby, добавление новых методов, 124
Ruby on Rails, фреймворк, 64

S

SAML, 64
schema, свойство, 642
scope, атрибут, 80, 449
script-interfaces, атрибут, 160
script-source, атрибут, 160
security, пространство имен, 436
send(), метод, 572, 704
Serializable, интерфейс, 480
server, свойство, 592, 606
serviceInterface, свойство, 486, 497,
583, 584
serviceUrl, свойство, 486, 498, 602
service, свойство, 584
SessionFactory, интерфейс, 272
SessionFactory, класс, 269
sessionFactory, свойство, 272, 303
Session, интерфейс, 268
setApplicationContext(), метод, 155
setAttribute(), метод, 604

setBeanFactory(), метод, 53, 155
setBeanName(), метод, 53, 154
setFrom(), метод, 704
setText(), метод, 704
setTo(), метод, 704
SimpleJaxWsServiceExporter,
класс, 500, 503
SimpleJdbcDaoSupport, класс, 248,
263
SimpleJdbcTemplate, класс, 49, 246,
258, 259, 263
SimpleMetadataStrategy, класс, 585
SimpleRemoteStatelessSessionProxy
FactoryBean, класс, 663
SimpleUrlHandlerMapping, 329
SingleConnectionDataSource,
класс, 252
SoapFaultMappingException-
Resolver, класс, 640
SourceHttpMessageConverter, 531
SpEL (Spring Expression Language),
язык выражений, 100
T(), оператор, 104
выборка элементов
коллекций, 114
доступ к элементам
коллекций, 112
и аннотации, 186
и аннотация @Value, 186
и типы, 104
литералы, 100
логические операции, 108
математические операции, 106
обработка коллекций, 111
операции, 105
отображение коллекций, 114
регулярные выражения, 110
сравнение значений, 107
ссылки по идентификаторам, 102
условные вычисления, 109
SpitterClientException,
исключение, 534
Spitter, пример, 259, 306
Spring
Spring AOP, 205
конфигурирование
в программном коде Java, 192
Spring AOP, 205

- в сравнении с AspectJ, 207
Spring Batch, фреймворк, 61
SpringBeanAutowiringSupport, класс, 501
Spring Dynamic Modules, фреймворк, 62
Spring Faces, фреймворк, 67
Spring Integration, фреймворк, 60
Spring JavaScript, библиотека, 67
Spring LDAP, фреймворк, 62
Spring Mobile, фреймворк, 61
Spring Modules, 284
Spring MVC, 321
 DispatcherServlet, 322
 Spring Web Flow, 393
 аннотации, 328
 запросы, 322
 настройка, 324
 обзор, 322
 обработка входных данных
 в контроллерах, 344
 создание контроллеров, 327
Spring MVC, фреймворк, 65
 поддержка REST, 66
Spring.NET, фреймворк, 63
Spring Rich Client, фреймворк, 63
Spring Roo, фреймворк, 64
Spring Security, 433
 Aegci Security, 433
 security, пространство имен, 436
 аспектно-ориентированное
 программирование, 433
 аутентификация
 с использованием LDAP, 457
 аутентификация с использова-
 нием базы данных, 455
 аутентификация с использова-
 нием репозитория в памяти, 453
 аутентификация средствами
 протокола HTTP, 442
 безопасность веб-запросов, 436
 внедрение зависимостей, 433
 защита методов, 463
 и язык выражений SpEL, 444
 конфигурационное пространство
 имен, 434, 435
 минимальная настройка
 безопасности, 438
 модули, 434
 обзор, 434
 поддержка функции
 «запомнить меня», 462
 принудительное использование
 протокола HTTPS, 446
 сервлет-фильтры, 437
 сравнение паролей
 с использованием LDAP, 459
 формы аутентификации, 439
 шифрование, 459
Spring Security, фреймворк, 60
 Kerberos, 64
 SAML, 64
Spring Social, фреймворк, 61
Spring Web Flow, 392
Spring Web Flow, фреймворк, 59
Spring Web Services, фреймворк, 60
Spring-WS, 613
 использование службы, 647
 маршалеры сообщений, 634, 636
 маршалеры сообщений
 на стороне клиента, 654
 настройка, 635
 обзор, 633
 отображение в конечные
 точки, 634
 отправка сообщений, 651
 преобразование исключений
 в ошибки SOAP, 634, 639
 развертывание службы, 646
 связывание, 632
 создание WSDL-файлов, 641, 644
 шаблоны веб-служб, 648
 шлюзы, 656
Spring, фреймворк
 JmsTemplate, 566
 MVC, фреймворк, 58
 Spring Batch, 61
 Spring Faces, 67
 Spring Integration, 60
 Spring JavaScript, 67
 Spring LDAP, 62
 Spring Mobile, 61
 Spring MVC, 65
 Spring.NET, 63
 Spring Rich Client, 63
 Spring Roo, 64

Spring Security, 60, 433
 Spring Social, 61
 Spring Web Flow, 59, 392
 Spring Web Services, 60
 аспектно-ориентированное программирование, 40, 46
 внедрение зависимостей, 35, 50
 внедрение компонентов
 Enterprise JavaBeans, 669
 дополнительные возможности, 59
 иерархия исключений JDBC в сравнении с иерархией исключений Spring, 242
 иерархия исключений, 241
 интеграция с фреймворком Hibernate, 265
 контейнер, 71
 контейнеры, 50, 56
 контекст приложения, 56, 343
 конфигурирование, 71
 модули, 55
 модуль AOP, 57
 модуль ORM, 57
 модуль доступа к данным, 57
 модуль поддержки тестирования, 59
 обмен сообщениями с помощью JMS, 556
 объявление компонентов, 70
 поддержка REST, 511
 поддержка транзакций, 300
 работа с базами данных, 238
 слои доступа к данным, 239
 смешивание с Enterprise JavaBeans (EJB), 660
 упрощение разработки на языке Java, 50
 устранение шаблонного кода, 47, 50
 фабрике компонентов, 56
 что нового в Spring 2.5?, 64
 что нового в Spring 3.0?, 65
SQLEception,
 исключение, 48, 241, 255
 SqlMapClientDaoSupport, класс, 248
 SqlMapClientTemplate, класс, 246

Stage, пример, 79
 StandardPBESStringEncryptor, класс, 691
 start-state, атрибут, 411
 StringHttpMessageConverter, 531, 542
 system-properties-mode, атрибут, 687

T

Tangosol Coherence, 285
 Tapestry, 34, 58
 target(), указатель для точки внедрения, 208
 then, атрибут, 401
 this(), указатель для точки внедрения, 208
 ticket, пример, 80
 TilesViewResolver, 335
 Tomcat, 249
 TopLinkJpaVendorAdapter, класс, 278
 toUpperCase(), метод, 104
 to, атрибут, 403
 TRACE, HTTP-метод, 519
 Trang, инструмент, 618
 TransactionCallback, интерфейс, 307
 TransactionDefinition, интерфейс, 310
 transactionManagerName, свойство, 306
 transactionmanager, атрибут, 319
 TransactionManager, интерфейс, 306
 TransactionTemplate, класс, 307
 Transaction, интерфейс, 304
 tx, пространство имен, 72, 309, 316
 types-matching, атрибут, 223
 type, атрибут, 191
 T(), оператор, 104

U

Unmarshaller, интерфейс, 636
 UnmarshallingFailureException, исключение, 641
 UnsupportedOperationException, исключение, 471
 URISyntaxException, исключение, 541

UrlBasedViewResolver, 335
url, атрибут, 460
URL в стиле RESTful, 514
use-expressions, атрибут, 444
username, свойство, 702
userPassword, атрибут, 459
user-search-base, атрибут, 458
user-search-filter, атрибут, 458
user-service, атрибут, 454
UserTransaction, интерфейс, 306
util, пространство имен, 72

V

ValidationFailureException, исключение, 641
value-ref, атрибут, 97
value, атрибут, 78, 86, 97, 101, 230, 522
var, атрибут, 448
VelocityConfigurer, компонент, 370
VelocityEngineFactoryBean, класс, 709
VelocityEngineUtils, класс, 710
VelocityEngine, класс, 709
VelocityLayoutViewResolver, 335
velocityProperties, свойство, 709
VelocityViewResolver, 335
VelocityViewResolver, компонент, 371
Velocity, механизм шаблонов, 368
настройка, 370
определение представлений, 369

разрешение представлений, 371
связывание полей форм, 374
форматирование дат и чисел, 372
viewResolvers, свойство, 527
view, атрибут, 399, 402
VisualVM, и JMX, 590

W

WEB-INF, каталог, 325
WebLogicJtaTransaction-Manager, 302
WebServiceGatewaySupport, класс, 657
WebServiceTemplate, класс, 647
WebSphere, 249
WebSphereUowTransaction-Manager, 302
WebWork, 34, 58
when, атрибут, 291
within(), указатель для точки внедрения, 208
wsdlDocumentUrl, свойство, 506

X

XmlAwareFormHttpMessage-Converter, 531
XML Schema (XSD), 618
XmlViewResolver, 335
XmlWebApplicationContext, 51
XML-сообщения, 616
XsltViewResolver, 336

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: 123242, Москва, а/я 20 или по электронному адресу: orders@aliants-kniga.ru.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: www.aliants-kniga.ru.

Оптовые закупки: тел. (499) 725-54-09, 725-50-27; электронный адрес books@aliants-kniga.ru.

Крейг Уоллс

Spring в действии

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru

Корректор *Синяева Г. И.*

Верстка *Чаннова А. А.*

Дизайн обложки *Мовчан А. Г.*

Подписано в печать 25.01.2013. Формат 60×90 1/16.

Гарнитура «Петербург». Печать офсетная.

Усл. печ. л. 47. Тираж 200 экз.

Веб-сайт издательства: www.dmk-press.ru

SPRING В ДЕЙСТВИИ

Фреймворк Spring Framework – необходимый инструмент для разработчиков приложений на Java, и Spring 3 несет в себе новые мощные особенности, такие как язык выражений SpEL, новые аннотации для работы с контейнером IoC и совершенно необходимая поддержка архитектуры REST. Эта книга является лучшим средством овладения Spring, как для тех, кто только открыл для себя этот фреймворк, так и для опытных пользователей Spring, желающих задействовать новые возможности версии 3.0.

Книга «Spring в действии, третье издание» продолжает традицию прагматичного подхода к освещению предмета обсуждения, заложенную в предыдущих изданиях, пользовавшихся большим спросом. Автор, Крейг Уоллс, обладает особым талантом придумывать весьма забавные примеры, сосредоточенные на особенностях и приемах их использования, которые действительно будут полезны. Это издание привлекает внимание к наиболее важным аспектам Spring 3.0, таким как поддержка REST, удаленные службы, обмен сообщениями, фреймворки: Security, MVC, Web Flow, и многое другое.

Внутри вы найдете обсуждение следующих тем:

- использования аннотаций для уменьшения объема конфигурации;
- работа с ресурсами RESTful;
- описание языка выражений Spring (Spring Expression Language, SpEL);
- фреймворки Security, Web Flow и другие.

Почти 100 000 разработчиков пользуются этой книгой, чтобы освоить Spring!

Интернет-магазин: www.dmk-press.ru
Книга – почтой: orders@aliants-kniga.ru
Оптовая продажа: “Альянс-книга”
тел.(499)725-54-09
books@aliants-kniga.ru



Единственная книга, которую я никому не буду давать – я слишком часто пользуюсь ею!

Джош Дэвис, Nokia

Де-факто – справочное руководство по фреймворку Spring.

Дэн Добрин, CIBC

Охватывает не только основы фреймворка Spring, но и расширенные его возможности.

*Чад Дэвис,
автор «Struts 2 in Action»*

Отличная книга от замечательного учителя.

*Роберт Хансон,
автор «GWT in Action»*

Прекрасная смесь юмора и технических сведений.

*Валентин Гремтаз,
Goomzee*

ISBN 978-5-94074-568-6



9 785940 745686 >