NAJLEPSZE RECEPTURY NA WYKORZYSTANIE SPRINGA!

Gary Mak, Josh Long, Daniel Rubio

Apress®

Tytuł oryginału: Spring Recipes: A Problem-Solution Approach

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-246-8226-3

Original edition copyright © 2010 by Gary Mak, Josh Long, and Daniel Rubio.
All rights reserved.

Polish edition copyright © 2014 by HELION SA.
All rights reserved.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/sprire.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/sprire
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- Kup książkę
- Poleć książkę
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

O autorach **13**

O recenzentach technicznych **15**

Podziękowania **17**

Wprowadzenie **19**

Dla kogo przeznaczona jest ta książka? 19
Struktura książki 20
Konwencje 21
Wymagania wstępne 21
Pobieranie kodu 21
Kontakt z autorami 21

Rozdział 1. Wprowadzenie do platformy Spring **23**

1.1. Tworzenie egzemplarza kontenera IoC w Springu 23
1.2. Konfigurowanie ziaren w kontenerze IoC 26
1.3. Tworzenie ziaren za pomocą konstruktora 34
1.4. Wybieranie konstruktora w przypadku wieloznaczności 37
1.5. Podawanie referencji do ziaren 39
1.6. Określanie typu danych elementów kolekcji 43
1.7. Tworzenie ziaren za pomocą interfejsu FactoryBean Springa 45
1.8. Definiowanie kolekcji za pomocą ziaren fabrycznych i schematu util 47
1.9. Sprawdzanie właściwości na podstawie zależności 49
1.10. Sprawdzanie właściwości z adnotacją @Required 51
1.11. Automatyczne łączenie ziaren za pomocą konfiguracji w pliku XML 53
1.12. Automatyczne łączenie ziaren z adnotacjami @Autowired i @Resource 57
1.13. Dziedziczenie konfiguracji ziarna 62
1.14. Skanowanie komponentów z parametru classpath 65
Podsumowanie 70

Rozdział 2. Zaawansowany kontener IoC w Springu **71**

2.1. Tworzenie ziaren za pomocą statycznych metod fabrycznych 71
2.2. Tworzenie ziaren za pomocą fabrycznej metody egzemplarza 72
2.3. Deklarowanie ziaren na podstawie pól statycznych 74
2.4. Deklarowanie ziaren na podstawie właściwości obiektów 75
2.5. Używanie języka wyrażeń dostępnego w Springu 77

Kup ksi k   Pole ksi k

2.6. Ustawianie zasięgu ziarna ... 82
2.7. Modyfikowanie procesu inicjowania i usuwania ziaren ... 84
2.8. Skracanie konfiguracji w XML-u za pomocą projektu Java Config ... 88
2.9. Ziarna znające zasoby kontenera ... 92
2.10. Wczytywanie zasobów zewnętrznych ... 93
2.11. Tworzenie postprocesorów ziaren ... 96
2.12. Zewnętrzne przechowywanie konfiguracji ziarna ... 99
2.13. Określanie komunikatów tekstowych ... 100
2.14. Komunikowanie się ze zdarzeniami aplikacji ... 102
2.15. Rejestrowanie edytorów właściwości w Springu ... 105
2.16. Tworzenie niestandardowych edytorów właściwości ... 108
2.17. Obsługa współbieżności za pomocą interfejsu TaskExecutor ... 109
Podsumowanie ... 117

Rozdział 3. Programowanie aspektowe i obsługa języka AspectJ w Springu ... 119
3.1. Włączanie obsługi adnotacji języka AspectJ w Springu ... 120
3.2. Deklarowanie aspektów za pomocą adnotacji języka AspectJ ... 122
3.3. Dostęp do informacji o punkcie złączenia ... 127
3.4. Określanie pierwszeństwa aspektów ... 128
3.5. Ponowne wykorzystanie definicji punktu przecięcia ... 130
3.6. Pisanie wyrażeń z punktami przecięcia w języku AspectJ ... 132
3.7. Dodawanie operacji do ziaren ... 136
3.8. Dodawanie stanu do ziarna ... 138
3.9. Deklarowanie aspektów za pomocą konfiguracji w XML-u ... 140
3.10. Wplatanie aspektów języka AspectJ w Springu w czasie ładowania ... 143
3.11. Konfigurowanie w Springu aspektów języka AspectJ ... 148
3.12. Wstrzykiwanie ziaren Springa do obiektów domenowych ... 149
Podsumowanie ... 152

Rozdział 4. Skrypty w Springu ... 153
4.1. Implementowanie ziaren za pomocą języków skryptowych ... 153
4.2. Wstrzykiwanie ziaren Springa do skryptów ... 157
4.3. Aktualizowanie ziaren ze skryptów ... 160
4.4. Wewnątrzwierszowe definiowanie kodu źródłowego skryptów ... 161
Podsumowanie ... 162

Rozdział 5. Bezpieczeństwo w Springu ... 163
5.1. Zabezpieczanie dostępu do adresów URL ... 163
5.2. Logowanie się do aplikacji sieciowych ... 172
5.3. Uwierzytelnianie użytkowników ... 176
5.4. Podejmowanie decyzji z zakresu kontroli dostępu ... 185
5.5. Zabezpieczanie wywołań metod ... 188
5.6. Obsługa zabezpieczeń w widokach ... 190
5.7. Zabezpieczanie obiektów domenowych ... 192
Podsumowanie ... 200

Rozdział 6. Integrowanie Springa z innymi platformami do tworzenia aplikacji sieciowych ... 203
6.1. Dostęp do Springa w dowolnych aplikacjach sieciowych ... 204
6.2. Używanie Springa w serwletach i filtrach ... 208
6.3. Integrowanie Springa z platformą Struts 1.x ... 212

Kup ksi k Pole ksi k

6.4. Integrowanie Springa z platformą JSF 218
6.5. Integrowanie Springa z platformą DWR 223
Podsumowanie 227

Rozdział 7. Platforma Spring Web Flow 229
7.1. Zarządzanie prostym przepływem sterowania za pomocą platformy Spring Web Flow 229
7.2. Modelowanie przepływów sterowania za pomocą różnych rodzajów stanów 236
7.3. Zabezpieczanie przepływów sterowania w aplikacjach sieciowych 247
7.4. Utrwalanie obiektów w przepływach sterowania w aplikacjach sieciowych 249
7.5. Integrowanie platformy Spring Web Flow z technologią JSF 255
7.6. Korzystanie z platformy RichFaces w platformie Spring Web Flow 262
Podsumowanie 266

Rozdział 8. Platforma Spring MVC 267
8.1. Tworzenie prostej aplikacji sieciowej za pomocą platformy Spring MVC 267
8.2. Wiązanie żądań za pomocą adnotacji @RequestMapping 278
8.3. Przechwytywanie żądań przy użyciu interceptorów przetwarzania 282
8.4. Określanie ustawień regionalnych użytkowników 285
8.5. Pliki zewnętrzne z tekstem dostosowanym do ustawień regionalnych 287
8.6. Określanie widoków na podstawie nazw 289
8.7. Widoki i negocjowanie treści 291
8.8. Wiązanie wyjątków z widokami 294
8.9. Przypisywanie wartości w kontrolerze za pomocą adnotacji @Value 296
8.10. Obsługiwanie formularzy za pomocą kontrolerów 297
8.11. Obsługa wielu formularzy za pomocą kontrolerów formularzy kreatora 310
8.12. Sprawdzanie poprawności ziaren za pomocą adnotacji (na podstawie standardu JSR-303) 319
8.13. Tworzenie widoków w formatach XLS i PDF 321
Podsumowanie 327

Rozdział 9. Usługi REST w Springu 329
9.1. Publikowanie usług typu REST w Springu 329
9.2. Dostęp do usług typu REST w Springu 333
9.3. Publikowanie danych z kanałów informacyjnych RSS i Atom 338
9.4. Publikowanie danych w formacie JSON w usługach typu REST 345
9.5. Dostęp do usług typu REST zwracających skomplikowane odpowiedzi w formacie XML 348
Podsumowanie 356

Rozdział 10. Spring i Flex 357
10.1. Wprowadzenie do środowiska Flex 359
10.2. Poza piaskownicę 364
10.3. Dodawanie obsługi narzędzia Spring BlazeDS Integration do aplikacji 374
10.4. Udostępnianie usług za pomocą technologii BlazeDS i Springa 378
10.5. Używanie obiektów działających po stronie serwera 384
10.6. Korzystanie z usług opartych na komunikatach w narzędziach BlazeDS i Spring 387
10.7. Wstrzykiwanie zależności w kliencie w ActionScripcie 398
Podsumowanie 402

Kup ksi k Pole ksi k

Rozdział 11. Grails ... 403
11.1. Pobieranie i instalowanie platformy Grails ... 403
11.2. Tworzenie aplikacji za pomocą platformy Grails ... 404
11.3. Wtyczki platformy Grails ... 408
11.4. Środowisko rozwojowe, produkcyjne i testowe w platformie Grails ... 410
11.5. Tworzenie klas domenowych aplikacji ... 412
11.6. Generowanie kontrolerów CRUD i widoków na potrzeby klas domenowych aplikacji ... 414
11.7. Właściwości związane z umiędzynarodawianiem komunikatów ... 417
11.8. Zmienianie systemu pamięci trwałej ... 420
11.9. Rejestrowanie danych wyjściowych ... 423
11.10. Przeprowadzanie testów jednostkowych i integracyjnych ... 425
11.11. Stosowanie niestandardowych układów i szablonów ... 430
11.12. Zapytania GORM ... 432
11.13. Tworzenie niestandardowych znaczników ... 434
Podsumowanie ... 436

Rozdział 12. Spring Roo ... 437
12.1. Konfigurowanie środowiska programistycznego pod kątem narzędzia Spring Roo ... 439
12.2. Tworzenie pierwszego projektu opartego na narzędziu Roo ... 441
12.3. Importowanie istniejących projektów do środowiska STS ... 446
12.4. Szybsze budowanie lepszych aplikacji ... 448
12.5. Usuwanie Roo z projektu ... 454
Podsumowanie ... 456

Rozdział 13. Testy w Springu ... 457
13.1. Tworzenie testów za pomocą platform JUnit i TestNG ... 458
13.2. Tworzenie testów jednostkowych i testów integracyjnych ... 463
13.3. Testy jednostkowe kontrolerów platformy Spring MVC ... 471
13.4. Zarządzanie kontekstem aplikacji w testach integracyjnych ... 472
13.5. Wstrzykiwanie konfiguracji testów w testach integracyjnych ... 478
13.6. Zarządzanie transakcjami w testach integracyjnych ... 482
13.7. Dostęp do bazy danych w testach integracyjnych ... 487
13.8. Stosowanie w Springu standardowych adnotacji związanych z testami ... 491
Podsumowanie ... 493

Rozdział 14. Platforma Spring Portlet MVC ... 495
14.1. Tworzenie prostego portletu za pomocą platformy Spring Portlet MVC ... 495
14.2. Wiązanie żądań kierowanych do portletów z metodami obsługi ... 503
14.3. Obsługa formularzy z portletów za pomocą prostych kontrolerów formularzy ... 510
Podsumowanie ... 517

Rozdział 15. Dostęp do danych ... 519
Problemy z bezpośrednim korzystaniem z JDBC ... 520
15.1. Używanie szablonu JDBC do aktualizowania bazy danych ... 526
15.2. Używanie szablonów JDBC do pobierania danych z bazy ... 530
15.3. Upraszczanie tworzenia szablonów JDBC ... 535
15.4. Używanie prostego szablonu JDBC w Javie 1.5 ... 537
15.5. Stosowanie nazwanych parametrów w szablonach JDBC ... 540
15.6. Obsługa wyjątków w platformie Spring JDBC ... 542

Kup ksi k Pole ksi k

15.7. Problemy z bezpośrednim używaniem platform do tworzenia odwzorowań ORM 547
15.8. Konfigurowanie fabryk zasobów ORM w Springu 556
15.9. Utrwalanie obiektów za pomocą szablonów ORM Springa 562
15.10. Utrwalanie obiektów za pomocą sesji kontekstowych platformy Hibernate 567
15.11. Utrwalanie obiektów za pomocą wstrzykiwania kontekstu w JPA 570
Podsumowanie 573

Rozdział 16. Zarządzanie transakcjami w Springu 575

16.1. Problemy z zarządzaniem transakcjami 576
16.2. Jak wybrać implementację menedżera transakcji? 581
16.3. Programowe zarządzanie transakcjami za pomocą interfejsu menedżera transakcji 583
16.4. Programowe zarządzanie transakcjami za pomocą szablonu transakcji 585
16.5. Deklaratywne zarządzanie transakcjami za pomocą rad transakcji 588
16.6. Deklaratywne zarządzanie transakcjami za pomocą adnotacji @Transactional 590
16.7. Ustawianie atrybutu propagation transakcji 591
16.8. Ustawianie atrybutu określającego poziom izolacji transakcji 596
16.9. Ustawianie atrybutu dotyczącego wycofywania transakcji 602
16.10. Ustawianie atrybutów związanych z limitem czasu i trybem tylko do odczytu 604
16.11. Zarządzanie transakcjami za pomocą wplatania w czasie ładowania 605
Podsumowanie 608

Rozdział 17. Ziarna EJB, zdalne wywołania i usługi sieciowe 609

17.1. Udostępnianie i wywoływanie usług za pomocą technologii RMI 609
17.2. Tworzenie komponentów EJB 2.x za pomocą Springa 613
17.3. Dostęp do dawnych komponentów EJB 2.x w Springu 618
17.4. Tworzenie komponentów EJB 3.0 w Springu 621
17.5. Dostęp do komponentów EJB 3.0 w Springu 623
17.6. Udostępnianie i wywoływanie usług za pomocą protokołu HTTP 625
17.7. Wybieranie sposobu tworzenia usług sieciowych SOAP 628
17.8. Udostępnianie i wywoływanie usług sieciowych SOAP z kontraktem pisanym na końcu za pomocą JAX-WS 630
17.9. Definiowanie kontraktu usługi sieciowej 636
17.10. Implementowanie usług sieciowych za pomocą platformy Spring-WS 640
17.11. Wywoływanie usług sieciowych za pomocą platformy Spring-WS 645
17.12. Tworzenie usług sieciowych za pomocą serializowania dokumentów XML 648
17.13. Tworzenie punktów końcowych usług za pomocą adnotacji 653
Podsumowanie 655

Rozdział 18. Spring w Javie EE 657

18.1. Eksportowanie ziaren Springa jako ziaren MBeans technologii JMX 657
18.2. Publikowanie i odbieranie powiadomień JMX 667
18.3. Dostęp do zdalnych ziaren MBeans technologii JMX w Springu 669
18.4. Wysyłanie e-maili za pomocą dostępnej w Springu obsługi poczty elektronicznej 672
18.5. Planowanie zadań za pomocą dostępnej w Springu obsługi Quartza 679
18.6. Planowanie zadań za pomocą przestrzeni nazw Scheduling ze Springa 3.0 683
Podsumowanie 686

Kup ksi k Pole ksi k

Rozdział 19. Komunikaty ... 687
19.1. Wysyłanie i pobieranie komunikatów JMS w Springu ... 688
19.2. Przekształcanie komunikatów JMS ... 698
19.3. Zarządzanie transakcjami JMS ... 700
19.4. Tworzenie w Springu obiektów POJO sterowanych komunikatami ... 701
19.5. Nawiązywanie połączeń ... 706
Podsumowanie ... 708

Rozdział 20. Platforma Spring Integration ... 709
20.1. Integrowanie jednego systemu z innym za pomocą EAI ... 710
20.2. Integrowanie dwóch systemów za pomocą technologii JMS ... 712
20.3. Wyszukiwanie informacji o kontekście w komunikatach platformy Spring Integration ... 716
20.4. Integrowanie dwóch systemów za pomocą systemu plików ... 718
20.5. Przekształcanie komunikatów z jednego typu na inny ... 720
20.6. Obsługa błędów za pomocą platformy Spring Integration ... 723
20.7. Rozdzielanie operacji w integrowanych mechanizmach — rozdzielacze i agregatory ... 726
20.8. Warunkowe przekazywanie za pomocą komponentu router ... 729
20.9. Dostosowywanie zewnętrznych systemów do magistrali ... 730
20.10. Podział zdarzeń na etapy za pomocą projektu Spring Batch ... 739
20.11. Używanie bram ... 740
Podsumowanie ... 745

Rozdział 21. Platforma Spring Batch ... 747
21.1. Konfigurowanie infrastruktury platformy Spring Batch ... 749
21.2. Odczyt i zapis ... 751
21.3. Tworzenie niestandardowych implementacji interfejsów ItemWriter i ItemReader ... 756
21.4. Przetwarzanie danych wejściowych przed ich zapisaniem ... 758
21.5. Łatwiejsza praca dzięki transakcjom ... 761
21.6. Ponawianie prób ... 762
21.7. Kontrolowanie wykonywania kroków ... 765
21.8. Uruchamianie zadania ... 769
21.9. Parametry w zadaniach ... 772
Podsumowanie ... 774

Rozdział 22. Przetwarzanie rozproszone w Springu ... 775
22.1. Przechowywanie stanu obiektów w klastrach za pomocą narzędzia Terracotta ... 777
22.2. Przetwarzanie w gridzie ... 785
22.3. Równoważenie obciążenia przy wykonywaniu metody ... 787
22.4. Przetwarzanie równoległe ... 790
22.5. Instalowanie aplikacji korzystających z narzędzia GridGain ... 792
Podsumowanie ... 796

Rozdział 23. Spring i jBPM ... 797
Procesy w oprogramowaniu ... 798
23.1. Modele przepływu pracy ... 800
23.2. Instalowanie systemu jBPM ... 802
23.3. Integrowanie systemu jBPM 4 ze Springiem ... 804

Kup książkę Poleć książkę

23.4. Tworzenie usług za pomocą Springa 809
23.5. Tworzenie procesu biznesowego 812
Podsumowanie 814

Rozdział 24. Spring i OSGi 815
24.1. Początki pracy z OSGi 816
24.2. Wprowadzenie do korzystania z platformy Spring Dynamic Modules 821
24.3. Eksportowanie usług za pomocą platformy Spring Dynamic Modules 825
24.4. Wyszukiwanie konkretnych usług w rejestrze OSGi 828
24.5. Publikowanie usług zgodnych z wieloma interfejsami 830
24.6. Dostosowywanie działania platformy Spring Dynamic Modules 831
24.7. Używanie serwera dm Server firmy SpringSource 832
24.8. Narzędzia firmy SpringSource 834
Podsumowanie 835

Skorowidz 837

Kup ksi k Pole ksi k

Kup ksi k

Pole ksi k

ROZDZIAŁ 15

Dostęp do danych

Z tego rozdziału dowiesz się, w jaki sposób Spring upraszcza zadania związane z dostępem do baz danych. Dostęp do danych jest niezbędny w większości aplikacji korporacyjnych. Zwykle potrzebny jest dostęp do danych zapisanych w bazach relacyjnych. Technologia JDBC jest ważną częścią Javy SE i obejmuje zestaw standardowych interfejsów API, które umożliwiają dostęp do relacyjnych baz danych niezależnie od ich producenta.

JDBC ma zapewniać interfejsy API, za pomocą których można wykonywać instrukcje SQL-a na bazach danych. Jednak przy stosowaniu JDBC trzeba samodzielnie zarządzać zasobami związanymi z bazą i obsługiwać wyjątki z tego obszaru. Aby ułatwić korzystanie z JDBC, w Springu udostępniono warstwę abstrakcyjną do komunikowania się z tą technologią. Szablony JDBC są ważnym elementem tej warstwy i mają udostępniać metody szablonowe używane do wykonywania różnego rodzaju operacji związanych z JDBC. Każda metoda szablonowa odpowiada za kontrolowanie ogólnego procesu działań i umożliwia przesłanianie poszczególnych zadań w jego ramach.

Jeśli interfejs JDBC nie spełnia Twoich wymagań lub uważasz, że w aplikacji warto zastosować technologię wyższego poziomu, może zainteresować Cię dostępna w Springu obsługa odwzorowań ORM. Z tego rozdziału dowiesz się, jak integrować platformy do tworzenia *odwzorowań obiektowo-relacyjnych* (ang. *Object/Relational Mapping* — ORM) z aplikacjami Springa. Spring obsługuje większość popularnych narzędzi tego typu, w tym Hibernate, JDO, iBATIS i JPA. Od wersji 3.0 Springa nie jest obsługiwana starsza wersja produktu TopLink, natomiast nadal obsługiwana jest oczywiście implementacja interfejsu JPA. Obsługa JPA jest jednak zróżnicowana. Obsługiwanych jest wiele implementacji JPA, w tym platforma Hibernate i wersje oparte na produkcie TopLink. W tym rozdziale koncentrujemy się na technologiach Hibernate i JPA. Jednak ponieważ obsługa narzędzi ORM w Springu jest spójna, techniki poznane w tym rozdziale będziesz mógł wykorzystać także dla innych narzędzi tego rodzaju.

ORM to nowa technologia utrwalania obiektów w relacyjnych bazach danych. Platformy do tworzenia odwzorowań ORM utrwalają obiekty na podstawie udostępnionych przez programistę odwzorowań (opartych na XML-u lub adnotacjach). Mogą to być odwzorowania między klasami i tabelami, właściwościami i kolumnami itd. Takie platformy w czasie wykonywania programu generują instrukcje SQL-a używane do utrwalania obiektów, dlatego nie musisz pisać instrukcji specyficznych dla bazy danych (chyba że chcesz wykorzystać funkcje dostępne w konkretnej bazie lub samodzielnie opracować zoptymalizowane polecenia). W efekcie aplikacja jest niezależna od bazy danych i w przyszłości będzie można ją łatwo dostosować do innej bazy. Platformy do tworzenia odwzorowań ORM pomagają znacznie ułatwić zapewnianie dostępu do danych w aplikacji w porównaniu z bezpośrednim stosowaniem JDBC.

Hibernate to otwarta i wydajna platforma do tworzenia odwzorowań ORM używana w społeczności skupionej wokół Javy. Hibernate obsługuje większość baz zgodnych z JDBC i korzysta z różnych dialektów, aby uzyskać dostęp do konkretnych baz danych. Oprócz podstawowych funkcji z zakresu odwzorowań ORM Hibernate udostępnia też zaawansowane rozwiązania, takie jak: pamięć podręczna, kaskada i odroczone wczytywanie. Dostępny jest też język zapytań HQL (ang. *Hibernate Query Language*), który pozwala pisać proste, ale dające duże możliwości zapytania obiektowe.

Kup ksi k

Pole ksi k

W JPA zdefiniowany jest zestaw standardowych adnotacji i interfejsów API umożliwiających utrwalanie obiektów zarówno w Javie SE, jak i w Javie EE. Interfejs JPA jest częścią specyfikacji EJB 3.0 i jest zdefiniowany w dokumencie JSR-220. JPA to tylko zestaw standardowych interfejsów API, dlatego wymaga zgodnego z JPA silnika wykonującego usługi z zakresu utrwalania obiektów. JPA można porównać do interfejsu JDBC, a silnik JPA — do sterownika JDBC. Hibernate można ustawić jako zgodny z JPA silnik za pomocą modułu Hibernate EntityManager. W tym rozdziale opisujemy przede wszystkim stosowanie JPA razem z platformą Hibernate. Wraz z Javą EE 6 pojawił się interfejs JPA 2.0. Spring obsługuje tę wersję, a od Springa 3.0.1 używana jest wersja 3.5 RC1 platformy Hibernate (jest to pierwsza wersja współdziałająca z JPA 2.0).

Problemy z bezpośrednim korzystaniem z JDBC

Załóżmy, że chcesz utworzyć aplikację przeznaczoną do rejestrowania samochodów. Aplikacja ma wykonywać podstawowe operacje CRUD na rekordach z danymi pojazdów. Rekordy mają być przechowywane w relacyjnej bazie danych i dostępne za pomocą interfejsu JDBC. Zacznij od zaprojektowania klasy `Vehicle`, która będzie reprezentować pojazdy w Javie.

```
package com.apress.springrecipes.vehicle;

public class Vehicle {

  private String vehicleNo;
  private String color;
  private int wheel;
  private int seat;

  // Konstruktory, gettery i settery
  ...
}
```

Konfigurowanie bazy danych aplikacji

Przed napisaniem aplikacji do rejestrowania pojazdów należy przygotować bazę danych. Ponieważ baza Apache Derby (*http://db.apache.org/derby/*) zużywa mało pamięci i jest łatwa do skonfigurowania, zastosowaliśmy właśnie ją. Derby to relacyjna baza danych o otwartym dostępie do kodu źródłowego dostępna na licencji Apache License i napisana w czystej Javie.

Derby może działać jako baza zagnieżdżona, a także w trybie klient-serwer. W trakcie testów lepszy jest tryb klient-serwer, ponieważ pozwala badać i modyfikować dane za pomocą dowolnych narzędzi z interfejsem graficznym korzystających z technologii JDBC. Jednym z takich narzędzi jest DTP (ang. *Data Tools Platform*) ze środowiska Eclipse.

■ **Uwaga** Aby uruchomić bazę Derby w trybie klient-serwer, uruchom skrypt `startNetworkScript` przeznaczony dla odpowiedniego systemu. Skrypt ten znajdziesz w katalogu *bin* instalacji bazy Derby.

Po uruchomieniu bazy Derby na komputerze lokalnym możesz połączyć się z nią za pomocą ustawień JDBC przedstawionych w tabeli 15.1.

Tabela 15.1. *Ustawienia JDBC potrzebne do połączenia się z bazą danych aplikacji*

Właściwość	**Wartość**
Klasa sterownika	`org.apache.derby.jdbc.ClientDriver`
Adres URL	`jdbc:derby://localhost:1527/vehicle;create=true`
Nazwa użytkownika	`app`
Hasło	`app`

Kup ksi k   Pole ksi k

■ **Uwaga** Potrzebny jest też sterownik klienta bazy Derby. Jeśli używasz Mavena, dodaj do projektu następującą zależność:

```
<dependency>
  <groupId>org.apache.derby</groupId>
  <artifactId>derbyclient</artifactId>
  <version>10.4.2.0</version>
</dependency>
```

Po nawiązaniu pierwszego połączenia z tą bazą zostanie utworzony egzemplarz `vehicle` bazy (jeśli wcześniej nie istniał). Jest tak, ponieważ w adresie URL znajduje się ustawienie `create=true`. Zauważ, że jeśli baza istnieje, to — zgodnie ze specyfikacją tego ustawienia — nie zostanie ponownie utworzona.

Aby połączyć się z bazą Derby, wykonaj następujące czynności:

1. Otwórz powłokę w systemie.
2. Wpisz polecenie `java –jar $DERBY_HOME/lib/derbyrun.jar ij` (w systemach uniksowych) lub `%DERBY_HOME%/lib/derbyrun.jar ij` (w systemie Windows).
3. Wywołaj instrukcję `CONNECT 'jdbc:derby://localhost:1527/vehicle;create=true';`.

Jako nazwę użytkownika i hasło możesz podać dowolne wartości, ponieważ w bazie Derby uwierzytelnianie jest domyślnie wyłączone. Następnie za pomocą poniższej instrukcji SQL-a utwórz tabelę `VEHICLE`. Posłuży ona do przechowywania rekordów z danymi pojazdów. Domyślnie tabela ta zostanie utworzona w schemacie bazy danych `APP`.

```
CREATE TABLE VEHICLE (
  VEHICLE_NO  VARCHAR(10) NOT NULL,
  COLOR       VARCHAR(10),
  WHEEL       INT,
  SEAT        INT,
  PRIMARY KEY (VEHICLE_NO)
);
```

Wzorzec projektowy DAO

Typowy błąd projektowy popełniany przez niedoświadczonych programistów polega na łączeniu kodu różnego typu (na przykład warstwy prezentacji, warstwy biznesowej i warstwy dostępu do danych) w jednym dużym module. To sprawia, że ponowne wykorzystanie kodu i jego pielęgnowanie jest utrudnione. Przyczyną jest ścisłe powiązanie komponentów. Ogólnym celem stosowania wzorca DAO (ang. *Data Access Object*) jest uniknięcie tych problemów w wyniku oddzielenia kodu odpowiedzialnego za dostęp do danych od warstwy biznesowej i warstwy prezentacji. Zgodnie z tym wzorcem warstwę dostępu do danych należy umieścić w niezależnych modułach — w obiektach DAO.

W aplikacji do rejestrowania pojazdów można utworzyć abstrakcyjną warstwę dostępu do danych z operacjami wstawiania, aktualizowania, usuwania i pobierania danych samochodów. Te operacje należy zadeklarować w interfejsie DAO, co pozwala zastosować różne technologie implementowania takiego interfejsu.

```
package com.apress.springrecipes.vehicle;

public interface VehicleDao {

  public void insert(Vehicle vehicle);
  public void update(Vehicle vehicle);
  public void delete(Vehicle vehicle);
  public Vehicle findByVehicleNo(String vehicleNo);
}
```

W większości interfejsów API JDBC zadeklarowane jest zgłaszanie wyjątków `java.sql.SQLException`. Jednak ponieważ tworzony tu interfejs ma obejmować tylko operacje dostępu do danych, nie powinien zależeć

Kup ksi k Pole ksi k

od żadnej technologii implementacji. Dlatego nie należy w ogólnym interfejsie deklarować zgłaszania specyficznych dla JDBC wyjątków `SQLException`. Często stosowana technika implementowania interfejsów DAO polega na umieszczeniu tego rodzaju wyjątku w wyjątku czasu wykonania (albo we własnej biznesowej klasie pochodnej od klasy `Exception`, albo w ogólnej klasie wyjątku).

Implementowanie interfejsu DAO za pomocą JDBC

Aby uzyskać dostęp do bazy przy użyciu JDBC, należy utworzyć implementację przedstawionego wcześniej interfejsu DAO `JdbcVehicleDao`. Implementacja interfejsu DAO musi nawiązać połączenie z bazą, by móc wykonywać instrukcje SQL-a. Można utworzyć takie połączenie za pomocą nazwy klasy sterownika, adresu URL bazy, nazwy użytkownika i hasła. Jednak w JDBC 2.0 i w nowszych wersjach można pobrać połączenie z bazą ze wstępnie skonfigurowanego obiektu `javax.sql.DataSource`. Nie wymaga to znajomości szczegółowych informacji o połączeniu.

```
package com.apress.springrecipes.vehicle;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

public class JdbcVehicleDao implements VehicleDao {

  private DataSource dataSource;

  public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
  }

  public void insert(Vehicle vehicle) {
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
               + "VALUES (?, ?, ?, ?)";
    Connection conn = null;
    try {
      conn = dataSource.getConnection();
      PreparedStatement ps = conn.prepareStatement(sql);
      ps.setString(1, vehicle.getVehicleNo());
      ps.setString(2, vehicle.getColor());
      ps.setInt(3, vehicle.getWheel());
      ps.setInt(4, vehicle.getSeat());
      ps.executeUpdate();
      ps.close();
    } catch (SQLException e) {
      throw new RuntimeException(e);
    } finally {
      if (conn != null) {
        try {
          conn.close();
        } catch (SQLException e) {}
      }
    }
  }

  public Vehicle findByVehicleNo(String vehicleNo) {
    String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
```


Kup ksi k Pole ksi k

```
    Connection conn = null;
    try {
      conn = dataSource.getConnection();
      PreparedStatement ps = conn.prepareStatement(sql);
      ps.setString(1, vehicleNo);

      Vehicle vehicle = null;
      ResultSet rs = ps.executeQuery();
      if (rs.next()) {
        vehicle = new Vehicle(rs.getString("VEHICLE_NO"),
          rs.getString("COLOR"), rs.getInt("WHEEL"),
          rs.getInt("SEAT"));
      }
      rs.close();
      ps.close();
      return vehicle;
    } catch (SQLException e) {
      throw new RuntimeException(e);
    } finally {
      if (conn != null) {
        try {
          conn.close();
        } catch (SQLException e) {}
      }
    }
  }

  public void update(Vehicle vehicle) {/* … */}

  public void delete(Vehicle vehicle) {/* … */}
}
```

Operacja wstawiania pojazdu to typowe aktualizowanie bazy za pomocą JDBC. Każde wywołanie przeznaczonej do tego metody powoduje pobranie połączenia ze źródła danych i wykonanie instrukcji SQL-a za pomocą tego połączenia. W utworzonym interfejsie DAO nie zadeklarowano zgłaszania żadnych sprawdzanych wyjątków, dlatego jeśli wystąpi wyjątek `SQLException`, należy umieścić go w niesprawdzanym wyjątku `RuntimeException`. Szczegółowe omówienie obsługi wyjątków w obiektach DAO znajdziesz w dalszej części rozdziału. Należy też pamiętać o zwolnieniu połączenia w bloku `finally`. Jeśli tego nie zrobisz, w aplikacji może zabraknąć połączeń.

Operacje aktualizowania i usuwania danych można pominąć, ponieważ technicznie są bardzo podobne do wstawiania danych. Jeśli chodzi o pobieranie danych, to oprócz wywołania odpowiedniej instrukcji SQL-a trzeba wyodrębnić dane ze zwróconego zbioru wyników i utworzyć obiekt typu `Vehicle`.

Konfigurowanie źródła danych w Springu

Interfejs `javax.sql.DataSource` to standardowy interfejs zdefiniowany w specyfikacji JDBC generujący obiekty typu `Connection`. Istnieje wiele implementacji tego interfejsu udostępnianych przez różne firmy i w rozmaitych projektach. Popularne są rozwiązania o otwartym dostępie do kodu źródłowego, C3PO i Apache Commons DBCP, jednak w większości serwerów aplikacji używane są udostępniane w nich implementacje. Ponieważ wszędzie implementowany jest ten sam interfejs `DataSource`, implementacje można łatwo zastępować innymi. W Javie Spring udostępnia kilka wygodnych, ale dających mniejsze możliwości implementacji. Najprostsza z nich to `DriverManagerDataSource`. Otwiera ona nowe połączenie za każdym razem, gdy otrzyma takie żądanie.

```
<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">
```


Kup ksi k Pole ksi k

```
  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
      value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url"
      value="jdbc:derby://localhost:1527/vehicle;create=true" />
    <property name="username" value="app" />
    <property name="password" value="app" />
  </bean>

  <bean id="vehicleDao"
    class="com.apress.springrecipes.vehicle.JdbcVehicleDao">
    <property name="dataSource" ref="dataSource" />
  </bean>
</beans>
```

Implementacja `DriverManagerDataSource` nie jest wydajna, ponieważ otwiera nowe połączenie za każdym razem, gdy klient tego zażąda. `SingleConnectionDataSource` to inna implementacja dostępna w Springu (jest to klasa pochodna od `DriverManagerDataSource`). Ta implementacja utrzymuje tylko jedno połączenie, które jest wielokrotnie wykorzystywane i stale otwarte. Oczywiście nie jest to dobre rozwiązanie w środowisku wielowątkowym.

Implementacje źródła danych dostępne w Springu stosuje się głównie w trakcie testów. Wiele produkcyjnych implementacji obsługuje pule połączeń. Na przykład moduł DBCP (ang. *Database Connection Pooling Services*) z biblioteki Apache Commons Library obejmuje kilka implementacji obsługujących pule połączeń. Spośród tych implementacji `BasicDataSource` przyjmuje te same właściwości połączeń co klasa `DriverManager` ↪`DataSource`, a ponadto umożliwia ustawienie początkowej liczby połączeń oraz maksymalnej liczby aktywnych połączeń w puli.

```
<bean id="dataSource"
  class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName"
    value="org.apache.derby.jdbc.ClientDriver" />
  <property name="url"
    value="jdbc:derby://localhost:1527/vehicle;create=true" />
  <property name="username" value="app" />
  <property name="password" value="app" />
  <property name="initialSize" value="2" />
  <property name="maxActive" value="5" />
</bean>
```

■ **Uwaga** Aby móc stosować implementacje źródła danych dostępne w module DBCP, trzeba je wskazać w parametrze `classpath`. Jeśli używasz Mavena, dodaj do projektu poniższą zależność:

```
<dependency>
  <groupId>commons-dbcp</groupId>
  <artifactId>commons-dbcp</artifactId>
  <version>1.2.1</version>
</dependency>
```

Wiele serwerów aplikacji Javy EE ma wbudowane implementacje źródła danych, które można skonfigurować z poziomu konsoli serwera lub w pliku konfiguracyjnym. Jeśli masz w serwerze aplikacji skonfigurowane źródło danych dostępne do wyszukiwania JNDI, możesz je znaleźć za pomocą ziarna `JndiObjectFactoryBean`.

```
<bean id="dataSource"
  class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="jdbc/VehicleDS" />
</bean>
```


Kup ksi k Pole ksi k

W Springu można uprościć wyszukiwanie JNDI przy użyciu elementu `jndi-lookup` zdefiniowanego w schemacie `jee`.

```
<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-3.0.xsd
    http://www.springframework.org/schema/jee
    http://www.springframework.org/schema/jee/spring-jee-3.0.xsd">

  <jee:jndi-lookup id="dataSource" jndi-name="jdbc/VehicleDS" />
  ...
</beans>
```

Uruchamianie obiektów DAO

Poniższa klasa `Main` testuje obiekt DAO. W tym celu używa go do wstawienia nowego pojazdu do bazy. Jeśli ten proces zakończy się powodzeniem, będzie można natychmiast pobrać dane pojazdu z bazy.

```
package com.apress.springrecipes.vehicle;

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

public class Main {

  public static void main(String[] args) {
    ApplicationContext context =
      new ClassPathXmlApplicationContext("beans.xml");

    VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
    Vehicle vehicle = new Vehicle("TEM0001", "Czerwony", 4, 4);
    vehicleDao.insert(vehicle);

    vehicle = vehicleDao.findByVehicleNo("TEM0001");
    System.out.println("Numer pojazdu: " + vehicle.getVehicleNo());
    System.out.println("Kolor: " + vehicle.getColor());
    System.out.println("Liczba kół: " + vehicle.getWheel());
    System.out.println("Liczba miejsc: " + vehicle.getSeat());
  }
}
```

Teraz możesz zaimplementować interfejs DAO bezpośrednio za pomocą JDBC. Jednak, jak widać w przedstawionej wcześniej implementacji, większość kodu JDBC wygląda podobnie i powtarza się dla każdej operacji na bazie danych. Ten zbędny kod sprawia, że metody obiektów DAO są długie i mało czytelne.

Następny krok

Inne podejście polega na wykorzystaniu narzędzi do tworzenia odwzorowań ORM. Ta technika pozwala napisać kod odwzorowujący encję z modelu domeny na tabelę bazy danych. Narzędzie ORM generuje następnie kod utrwalający dane z klasy w bazie. Uwalnia to programistę od wielu zadań. Dzięki temu wystarczy napisać kod biznesowy i model domeny — nie trzeba martwić się o parser SQL-a z bazy danych. Jednak programista rezygnuje tym samym z zachowania pełnej kontroli nad komunikacją między klientem a bazą i musi zaufać, że warstwa ORM będzie działać w poprawny sposób.

Kup ksi k Pole ksi k

15.1. Używanie szablonu JDBC do aktualizowania bazy danych

Problem

Używanie JDBC jest żmudne i często prowadzi do dodawania zbędnych wywołań interfejsu API, podczas gdy wiele zadań mogłoby zostać wykonanych automatycznie. Na przykład aby zaimplementować operację aktualizowania za pomocą JDBC, trzeba wykonać poniższe czynności (przy czym większość z nich jest zbędna):

1. Pobieranie połączenia z bazą ze źródła danych.
2. Tworzenie obiektu typu `PreparedStatement` na podstawie tego połączenia.
3. Wiązanie parametrów z obiektem typu `PreparedStatement`.
4. Wykonywanie instrukcji z obiektu typu `PreparedStatement`.
5. Obsługiwanie wyjątków `SQLException`.
6. Porządkowanie obiektu z instrukcją i połączenia.

JDBC to bardzo niskopoziomowy interfejs API, jednak dzięki szablonom JDBC można zwiększyć produktywność (mniej czasu potrzeba wtedy na wykonywanie zadań pomocniczych, a więcej — na pisanie logiki aplikacji) i uprościć bezpieczne korzystanie z tego interfejsu.

Rozwiązanie

W klasie `org.springframework.jdbc.core.JdbcTemplate` zadeklarowanych jest wiele przeciążonych metod szablonowych `update()`, które pozwalają kontrolować proces aktualizowania danych. Poszczególne wersje tej metody umożliwiają przesłanianie różnych podzbiorów zadań z domyślnego procesu. W platformie Spring JDBC wstępnie zdefiniowanych jest kilka interfejsów wywołań zwrotnych, które dotyczą różnych podzbiorów zadań. Możesz zaimplementować jeden z takich interfejsów i przekazać obiekt tego typu do odpowiedniej metody `update()` w celu wykonania procesu.

Jak to działa?

Aktualizowanie bazy danych za pomocą kreatora instrukcji

Pierwszy z omawianych tu interfejsów wywołań zwrotnych to `PreparedStatementCreator`. Możesz zaimplementować ten interfejs, aby przesłonić operacje tworzenia instrukcji (krok 2.) i wiązania parametrów (krok 3.) ogólnego procesu aktualizowania danych. Jeśli chcesz wstawiać pojazdy do bazy, zaimplementuj interfejs `PreparedStatementCreator` w następujący sposób:

```
package com.apress.springrecipes.vehicle;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import org.springframework.jdbc.core.PreparedStatementCreator;

public class InsertVehicleStatementCreator implements PreparedStatementCreator {

  private Vehicle vehicle;

  public InsertVehicleStatementCreator(Vehicle vehicle) {
    this.vehicle = vehicle;
  }
```


Kup ksi k Pole ksi k

```
  public PreparedStatement createPreparedStatement(Connection conn)
    throws SQLException {
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
               + "VALUES (?, ?, ?, ?)";
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setString(1, vehicle.getVehicleNo());
    ps.setString(2, vehicle.getColor());
    ps.setInt(3, vehicle.getWheel());
    ps.setInt(4, vehicle.getSeat());
    return ps;
  }
}
```

W implementacji interfejsu `PreparedStatementCreator` połączenie z bazą danych jest przekazywane jako argument metody `createPreparedStatement()`. W tej metodzie wystarczy utworzyć obiekt typu `PreparedStatement` na podstawie tego połączenia i powiązać parametry z tym obiektem. Metoda ta powinna zwracać obiekt typu `PreparedStatement`. Zauważ, że w sygnaturze tej metody zadeklarowane jest zgłaszanie wyjątku `SQLException`, co oznacza, że nie musisz samodzielnie obsługiwać wyjątków tego typu.

Teraz możesz wykorzystać kreator instrukcji do uproszczenia operacji wstawiania pojazdów do bazy. Przede wszystkim należy utworzyć szablon (obiekt typu `JdbcTemplate`) i przekazać dla tego szablonu źródło danych, które posłuży do otrzymania połączenia. Następnie wystarczy wywołać metodę `update()` i przekazać do niej przeznaczony dla szablonu kreator instrukcji, aby zakończyć proces wstawiania danych.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
  ...
  public void insert(Vehicle vehicle) {
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
    jdbcTemplate.update(new InsertVehicleStatementCreator(vehicle));
  }
}
```

Jeśli interfejs `PreparedStatementCreator` lub inne interfejsy wywołań zwrotnych są używane tylko w jednej metodzie, zwykle lepiej jest implementować je jako klasy wewnętrzne. Jest tak, ponieważ można wtedy uzyskać dostęp do zmiennych lokalnych i argumentów metody bezpośrednio w klasie wewnętrznej, zamiast przekazywać te dane w argumentach konstruktora. Jedyne ograniczenie polega tu na tym, że takie zmienne i argumenty muszą mieć modyfikator `final`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;

public class JdbcVehicleDao implements VehicleDao {
  ...
  public void insert(final Vehicle vehicle) {
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

    jdbcTemplate.update(new PreparedStatementCreator() {
      public PreparedStatement createPreparedStatement(Connection conn)
        throws SQLException {
        String sql = "INSERT INTO VEHICLE "
                   + "(VEHICLE_NO, COLOR, WHEEL, SEAT) "
                   + "VALUES (?, ?, ?, ?)";
        PreparedStatement ps = conn.prepareStatement(sql);
```


Kup ksi k Pole ksi k

```
        ps.setString(1, vehicle.getVehicleNo());
        ps.setString(2, vehicle.getColor());
        ps.setInt(3, vehicle.getWheel());
        ps.setInt(4, vehicle.getSeat());
        return ps;
      }
    });
  }
}
```

Teraz możesz usunąć klasę `InsertVehicleStatementCreator`, ponieważ nie będzie już potrzebna.

Aktualizowanie bazy danych za pomocą settera instrukcji

Drugi interfejs wywołań zwrotnych, `PreparedStatementSetter`, odpowiada tylko za wiązanie parametrów (krok 3.) z ogólnego procesu aktualizowania bazy.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementSetter;

public class JdbcVehicleDao implements VehicleDao {
  ...
  public void insert(final Vehicle vehicle) {
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
             + "VALUES (?, ?, ?, ?)";
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

    jdbcTemplate.update(sql, new PreparedStatementSetter() {

      public void setValues(PreparedStatement ps)
        throws SQLException {
        ps.setString(1, vehicle.getVehicleNo());
        ps.setString(2, vehicle.getColor());
        ps.setInt(3, vehicle.getWheel());
        ps.setInt(4, vehicle.getSeat());
      }
    });
  }
}
```

Inna wersja metody szablonowej `update()` przyjmuje jako argumenty instrukcję SQL-a i obiekt typu `PreparedStatementSetter`. Ta wersja metody tworzy obiekt typu `PreparedStatement` na podstawie podanej instrukcji SQL-a. Programista używający tego interfejsu musi tylko powiązać parametry z tym obiektem.

Aktualizowanie bazy danych za pomocą instrukcji SQL-a i wartości parametrów

Najprostsza wersja metody `update()` przyjmuje instrukcję SQL-a i tablicę obiektów z parametrami instrukcji. Ta wersja tworzy obiekt typu `PreparedStatement` na podstawie podanej instrukcji SQL-a i automatycznie wiąże parametry. Dlatego nie musisz przesłaniać żadnego etapu procesu aktualizowania bazy.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
  ...
  public void insert(final Vehicle vehicle) {
```


Kup ksi k   Pole ksi k

```
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
               + "VALUES (?, ?, ?, ?)";
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

    jdbcTemplate.update(sql, new Object[] { vehicle.getVehicleNo(),
      vehicle.getColor(),vehicle.getWheel(), vehicle.getSeat() });
  }
}
```

Spośród trzech opisanych wersji metody `update()` ostatnia jest najprostsza, ponieważ nie wymaga implementowania żadnych interfejsów wywołań zwrotnych. Ponadto można zrezygnować ze wszystkich metod w formacie `setX` (`setInt`, `setString` itd.) przy ustawianiu parametrów zapytania. Za to najwięcej możliwości daje pierwsza z przedstawionych metod, ponieważ pozwala wykonać wstępne przetwarzanie obiektu typu `PreparedStatement` przed wykonaniem instrukcji. W praktyce należy zawsze wybierać najprostszą wersję, która spełnia wymagania programisty.

Klasa `JdbcTemplate` udostępnia też inne przeciążone wersje metody `update()`. Więcej informacji na ten temat znajdziesz w dokumentacji Javadoc.

Masowe aktualizowanie bazy danych

Załóżmy, że chcesz wstawić do bazy zbiór pojazdów. Jeśli wielokrotnie wywołasz metodę `insert()`, proces aktualizacji potrwa bardzo długo, ponieważ trzeba będzie za każdym razem kompilować instrukcję SQL-a. Dlatego lepiej jest dodać do interfejsu DAO nową metodę, służącą do wstawiania zbioru pojazdów.

```
package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
  ...
  public void insertBatch(List<Vehicle> vehicles);
}
```

Klasa `JdbcTemplate` udostępnia metodę `batchUpdate()`. Służy ona do masowego aktualizowania bazy. Argumentami tej metody są instrukcja SQL-a i obiekt typu `BatchPreparedStatementSetter`. W tej metodzie instrukcja jest kompilowana (przygotowywana) tylko raz, a następnie wielokrotnie wykonywana. Jeśli sterownik bazy danych obsługuje wersję 2.0 JDBC, wspomniana metoda automatycznie wykorzystuje mechanizm aktualizowania masowego, co pozwala zwiększyć wydajność aplikacji.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BatchPreparedStatementSetter;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
  ...
  public void insertBatch(final List<Vehicle> vehicles) {
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
               + "VALUES (?, ?, ?, ?)";
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

    jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {

      public int getBatchSize() {
        return vehicles.size();
      }

      public void setValues(PreparedStatement ps, int i)
        throws SQLException {
        Vehicle vehicle = vehicles.get(i);
        ps.setString(1, vehicle.getVehicleNo());
```


Kup ksi k Pole ksi k

```
        ps.setString(2, vehicle.getColor());
        ps.setInt(3, vehicle.getWheel());
        ps.setInt(4, vehicle.getSeat());
      }
    });
  }
}
```

Wstawianie masowe możesz przetestować za pomocą poniższego fragmentu kodu z klasy `Main`:

```
package com.apress.springrecipes.vehicle;
...
public class Main {

  public static void main(String[] args) {
    ...
    VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
    Vehicle vehicle1 = new Vehicle("TEM0002", "Niebieski", 4, 4);
    Vehicle vehicle2 = new Vehicle("TEM0003", "Czarny", 4, 6);
    vehicleDao.insertBatch(
      Arrays.asList(new Vehicle[] { vehicle1, vehicle2 }));
  }
}
```

15.2. Używanie szablonów JDBC do pobierania danych z bazy

Problem

Aby zaimplementować operację pobierania danych za pomocą JDBC, trzeba wykonać opisane poniżej czynności. Dwie z nich (kroki 5. i 6.) dodano w porównaniu z operacją aktualizowania bazy.

1. Pobieranie połączenia z bazą ze źródła danych.
2. Tworzenie obiektu typu `PreparedStatement` na podstawie tego połączenia.
3. Wiązanie parametrów z obiektem typu `PreparedStatement`.
4. Wykonywanie instrukcji z obiektu typu `PreparedStatement`.
5. Przechodzenie po zwróconym zbiorze wyników.
6. Pobieranie danych ze zbioru wyników.
7. Obsługiwanie wyjątków `SQLException`.
8. Porządkowanie obiektu z instrukcją i połączenia.

Jedyne etapy związane z logiką biznesową to definicja zapytania i pobieranie wyników z ich zbioru. Obsługę pozostałych kroków lepiej jest przekazać szablonowi JDBC.

Rozwiązanie

W klasie `JdbcTemplate` zadeklarowane są liczne przeciążone metody szablonowe `query()`, które pozwalają kontrolować ogólny proces pobierania danych. Podobnie jak przy aktualizowaniu bazy możesz za pomocą interfejsów `PreparedStatementCreator` i `PreparedStatementSetter` przesłonić tworzenie instrukcji (krok 2.) i wiązanie parametrów (krok 3.). Ponadto dostępne w Springu mechanizmy do obsługi JDBC oferują wiele sposobów modyfikowania etapu pobierania danych (krok 6.).

Kup ksi k   Pole ksi k

Jak to działa?

Pobieranie danych za pomocą interfejsu RowCallbackHandler

`RowCallbackHandler` to podstawowy interfejs umożliwiający przetwarzanie bieżącego wiersza zbioru wyników. Jedna z wersji metody `query()` przechodzi po zbiorze wyników i wywołuje dla każdego z nich metodę z tego interfejsu. W efekcie metoda `processRow()` jest wywoływana jednokrotnie dla każdego wiersza ze zwróconego zbioru wyników.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowCallbackHandler;

public class JdbcVehicleDao implements VehicleDao {
  ...
  public Vehicle findByVehicleNo(String vehicleNo) {
    String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

    final Vehicle vehicle = new Vehicle();
    jdbcTemplate.query(sql, new Object[] { vehicleNo },
      new RowCallbackHandler() {
        public void processRow(ResultSet rs) throws SQLException {
          vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
          vehicle.setColor(rs.getString("COLOR"));
          vehicle.setWheel(rs.getInt("WHEEL"));
          vehicle.setSeat(rs.getInt("SEAT"));
        }
      });
    return vehicle;
  }
}
```

Ponieważ to zapytanie SQL-a zwraca maksymalnie jeden wiersz, możesz utworzyć obiekt `vehicle` jako zmienną lokalną i ustawić jego właściwości na podstawie danych pobranych ze zbioru wyników. Gdy zbiór wyników obejmuje więcej wierszy, najpierw trzeba zapisać obiekty na liście.

Pobieranie danych za pomocą interfejsu RowMapper

Interfejs `RowMapper<T>` jest bardziej ogólny niż `RowCallbackHandler`. Pozwala odwzorować jeden wiersz ze zbioru wyników na określony obiekt. Dlatego można go zastosować zarówno do jedno-, jak i wielowierszowych zbiorów wyników. Jeśli chodzi o wielokrotne wykorzystanie kodu, lepiej jest zaimplementować interfejs `RowMapper<T>` jako zwykłą klasę niż jako klasę wewnętrzną. W metodzie `mapRow()` tego interfejsu trzeba utworzyć reprezentujący wiersz obiekt, który później jest zwracany.

```
package com.apress.springrecipes.vehicle;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class VehicleRowMapper implements RowMapper<Vehicle> {

  public Vehicle mapRow(ResultSet rs, int rowNum) throws SQLException {
    Vehicle vehicle = new Vehicle();
    vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
```


Kup ksi k Pole ksi k

```
    vehicle.setColor(rs.getString("COLOR"));
    vehicle.setWheel(rs.getInt("WHEEL"));
    vehicle.setSeat(rs.getInt("SEAT"));
    return vehicle;
  }
}
```

Jak wspomniano, interfejs RowMapper<T> można wykorzystać zarówno do jedno-, jak i wielowierszowych zbiorów wyników. Gdy pobierasz unikatowy obiekt, tak jak w metodzie findByVehicleNo(), musisz wywołać metodę queryForObject() klasy JdbcTemplate.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
  ...
  public Vehicle findByVehicleNo(String vehicleNo) {
    String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

    Vehicle vehicle = (Vehicle) jdbcTemplate.queryForObject(sql,
      new Object[] { vehicleNo }, new VehicleRowMapper());
    return vehicle;
  }
}
```

Spring udostępnia wygodną implementację interfejsu RowMapper<T>, BeanPropertyRowMapper<T>, która potrafi automatycznie odwzorować wiersz na nowy obiekt określonej klasy. Zauważ, że używana klasa musi być klasą z najwyższego poziomu zawierającą konstruktor domyślny (bezargumentowy). Wspomniana implementacja najpierw tworzy obiekt tej klasy, a następnie odwzorowuje wartość z każdej kolumny na wartość właściwości o pasującej nazwie. Ta technika obsługuje dopasowywanie nazw właściwości (na przykład vehicleNo) do identycznych nazw kolumn lub podobnych nazw z podkreśleniami (na przykład VEHICLE_NO).

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {

  ...
  public Vehicle findByVehicleNo(String vehicleNo) {
    String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
    BeanPropertyRowMapper<Vehicle> vehicleRowMapper =
      BeanPropertyRowMapper.newInstance(Vehicle.class);
    Vehicle vehicle = getSimpleJdbcTemplate().queryForObject(
      sql, vehicleRowMapper, vehicleNo);
    return vehicle;
  }
}
```

Pobieranie wielu wierszy

Teraz przyjrzyj się temu, jak pobierać zbiory wyników z wieloma wierszami. Załóżmy, że potrzebujesz w interfejsie DAO metody findAll() do pobierania wszystkich pojazdów.

```
package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
```


Kup ksi k Pole ksi k

```
  ...
  public List<Vehicle> findAll();
}
```

Gdy interfejs `RowMapper<T>` nie jest używany, można wywołać metodę `queryForList()` i przekazać do niej instrukcję SQL-a. Zwrócony wynik to lista odwzorowań. Każde z nich obejmuje wiersz zbioru wynikowego, przy czym kluczami są nazwy kolumn.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
  ...
  public List<Vehicle> findAll() {
    String sql = "SELECT * FROM VEHICLE";
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

    List<Vehicle> vehicles = new ArrayList<Vehicle>();
    List<Map<String,Object>> rows = jdbcTemplate.queryForList(sql);
    for (Map<String, Object> row : rows) {
      Vehicle vehicle = new Vehicle();
      vehicle.setVehicleNo((String) row.get("VEHICLE_NO"));
      vehicle.setColor((String) row.get("COLOR"));
      vehicle.setWheel((Integer) row.get("WHEEL"));
      vehicle.setSeat((Integer) row.get("SEAT"));
      vehicles.add(vehicle);
    }
    return vehicles;
  }
}
```

Metodę `findAll()` możesz przetestować za pomocą poniższego fragmentu kodu z klasy `Main`.

```
package com.apress.springrecipes.vehicle;
...
public class Main {

  public static void main(String[] args) {
    ...
    VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
    List<Vehicle> vehicles = vehicleDao.findAll();
    for (Vehicle vehicle : vehicles) {
      System.out.println("Numer pojazdu: " + vehicle.getVehicleNo());
      System.out.println("Kolor: " + vehicle.getColor());
      System.out.println("Liczba kół: " + vehicle.getWheel());
      System.out.println("Liczba miejsc: " + vehicle.getSeat());
    }
  }
}
```

Jeśli używasz obiektu typu `RowMapper<T>` do odwzorowania wierszy ze zbioru wyników, listę odwzorowanych obiektów możesz pobrać za pomocą metody `query()`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
  ...
```


Kup ksi k Pole ksi k

```
  public List<Vehicle> findAll() {
    String sql = "SELECT * FROM VEHICLE";
    RowMapper<Vehicle> rm =
      BeanPropertyRowMapper.newInstance(Vehicle.class);
    List<Vehicle> vehicles = getSimpleJdbcTemplate().query(sql, rm);
    return vehicles;
  }
}
```

Pobieranie pojedynczych wartości

Na zakończenie przyjrzyj się zapytaniom dotyczącym jednowierszowych i jednokolumnowych zbiorów wyników. Dodaj do interfejsu DAO poniższe operacje:

```
package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
  ...
  public String getColor(String vehicleNo);
  public int countAll();
}
```

Aby pobrać jeden łańcuch znaków, można wywołać przeciążoną metodę `queryForObject()`. Przyjmuje ona argument typu `java.lang.Class`. Ta metoda pomaga odwzorować wynikową wartość na określony typ. Dla wartości całkowitoliczbowych można wywołać wygodną metodę `queryForInt()`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
  ...

  public String getColor(String vehicleNo) {
    String sql = "SELECT COLOR FROM VEHICLE WHERE VEHICLE_NO = ?";
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

    String color = (String) jdbcTemplate.queryForObject(sql,
      new Object[] { vehicleNo }, String.class);
    return color;
  }

  public int countAll() {
    String sql = "SELECT COUNT(*) FROM VEHICLE";
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

    int count = jdbcTemplate.queryForInt(sql);
    return count;
  }
}
```

Obie wymienione metody można przetestować za pomocą poniższego fragmentu kodu z klasy `Main`:

```
package com.apress.springrecipes.vehicle;
...
public class Main {

  public static void main(String[] args) {
    ...
    VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
```


Kup ksi   k   Pole   ksi   k

```
    int count = vehicleDao.countAll();
    System.out.println("Liczba pojazdów: " + count);
    String color = vehicleDao.getColor("TEM0001");
    System.out.println("Kolor pojazdu [TEM0001]: " + color);
  }
}
```

15.3. Upraszczanie tworzenia szablonów JDBC

Problem

Tworzenie nowego obiektu typu `JdbcTemplate` za każdym razem, gdy jest potrzebny, to niewydajne rozwiązanie, ponieważ trzeba powtarzać instrukcje i ponosić koszty generowania nowego obiektu.

Rozwiązanie

Klasa `JdbcTemplate` jest bezpieczna ze względu na wątki, dlatego można zadeklarować jeden obiekt tego typu w kontenerze IoC i wstrzykiwać ten egzemplarz do wszystkich obiektów DAO. Ponadto platforma Spring JDBC udostępnia wygodną klasę `org.springframework.jdbc.core.support.JdbcDaoSupport`, która pomaga uprościć implementowanie interfejsu DAO. W tej klasie zadeklarowana jest właściwość `jdbcTemplate`, której wartość można wstrzyknąć z kontenera IoC lub utworzyć automatycznie na podstawie źródła danych (na przykład `JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource)`). Implementację DAO można utworzyć jako klasę pochodną od `JdbcDaoSupport`, dzięki czemu wspomniana właściwość zostanie odziedziczona.

Jak to działa?

Wstrzykiwanie szablonu JDBC

Do tej pory w każdej metodzie interfejsu DAO tworzyłeś nowy obiekt typu `JdbcTemplate`. Jednak możesz też wstrzykiwać go na poziomie klasy i we wszystkich metodach DAO używać tego wstrzykniętego obiektu. Dla uproszczenia w poniższym kodzie prezentujemy tylko zmiany w metodzie `insert()`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {

  private JdbcTemplate jdbcTemplate;

  public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

  public void insert(final Vehicle vehicle) {
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";

    jdbcTemplate.update(sql, new Object[] { vehicle.getVehicleNo(),
      vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
  }
  ...
}
```


Kup ksi k Pole ksi k

Szablon JDBC wymaga ustawienia źródła danych. Możesz wstrzyknąć wartość właściwości albo za pomocą settera, albo przy użyciu argumentu konstruktora. Następnie można wstrzyknąć szablon JDBC do obiektu DAO.

```
<beans ...>
  ...
  <bean id="jdbcTemplate"
    class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource" />
  </bean>

  <bean id="vehicleDao"
    class="com.apress.springrecipes.vehicle.JdbcVehicleDao">
    <property name="jdbcTemplate" ref="jdbcTemplate" />
  </bean>
</beans>
```

Tworzenie klasy pochodnej od JdbcDaoSupport

Klasa `org.springframework.jdbc.core.support.JdbcDaoSupport` udostępnia metody `setDataSource()` i `setJdbcTemplate()`. Można utworzyć pochodną od niej klasę DAO, aby zapewnić dziedziczenie tych metod. Następnie można albo bezpośrednio wstrzyknąć szablon JDBC, albo wstrzyknąć źródło danych i na jego podstawie utworzyć taki szablon. Poniższy fragment kodu pochodzi z klasy `JdbcDaoSupport` Springa:

```
package org.springframework.jdbc.core.support;
...
public abstract class JdbcDaoSupport extends DaoSupport {

  private JdbcTemplate jdbcTemplate;

  public final void setDataSource(DataSource dataSource) {
    if( this.jdbcTemplate == null || dataSource != this.jdbcTemplate.
      ➥getDataSource() ){
      this.jdbcTemplate = createJdbcTemplate(dataSource);
      initTemplateConfig();
    }
  }
  ...
  public final void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
    initTemplateConfig();
  }

  public final JdbcTemplate getJdbcTemplate() {
    return this.jdbcTemplate;
  }
  ...
}
```

Aby pobrać szablon JDBC, wystarczy w metodach klasy DAO wywołać metodę `getJdbcTemplate()`. W klasie DAO należy też usunąć właściwości `dataSource` i `jdbcTemplate` oraz powiązane z nimi settery, ponieważ teraz te składowe są dziedziczone. Dla uproszczenia poniżej przedstawiamy tylko zmiany potrzebne w metodzie `insert()`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.support.JdbcDaoSupport;

public class JdbcVehicleDao extends JdbcDaoSupport implements VehicleDao {

  public void insert(final Vehicle vehicle) {
```


Kup ksi k   Pole ksi k

```
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
               + "VALUES (?, ?, ?, ?)";
    getJdbcTemplate().update(sql, new Object[] { vehicle.getVehicleNo(),
      vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
  }
  ...
}
```

Ponieważ teraz klasa DAO jest pochodna od klasy `JdbcDaoSupport`, dziedziczy metodę `setDataSource()`. Możesz więc wstrzyknąć źródło danych do obiektu DAO, aby utworzyć szablon JDBC.

```
<beans ...>
  ...
  <bean id="vehicleDao"
    class="com.apress.springrecipes.vehicle.JdbcVehicleDao">
    <property name="dataSource" ref="dataSource" />
  </bean>
</beans>
```

15.4. Używanie prostego szablonu JDBC w Javie 1.5

Problem

Klasa `JdbcTemplate` w większości sytuacji działa dobrze, jednak można zastosować lepsze rozwiązanie, aby wykorzystać możliwości Javy 1.5.

Rozwiązanie

Klasa `org.springframework.jdbc.core.simple.SimpleJdbcTemplate` to zmodyfikowana wersja klasy `JdbcTemplate`, wykorzystująca upraszczające pracę mechanizmy Javy 1.5 — na przykład automatyczne traktowanie typów prostych jak obiektów (ang. *autoboxing*), typy generyczne i argumenty na liście o zmiennej długości.

Jak to działa?

Używanie klasy SimpleJdbcTemplate do aktualizowania baz danych

Wiele metod klasycznej klasy `JdbcTemplate` wymaga przekazywania parametrów instrukcji za pomocą tablicy obiektów. W klasie `SimpleJdbcTemplate` parametry można przekazywać na zmiennej długości liście argumentów. Dzięki temu nie musisz zapisywać ich w tablicy. Aby zastosować klasę `SimpleJdbcTemplate`, możesz bezpośrednio utworzyć obiekt tego typu lub pobrać go w wyniku utworzenia klasy pochodnej od klasy `SimpleJdbcDaoSupport`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
  VehicleDao {

  public void insert(Vehicle vehicle) {
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
               + "VALUES (?, ?, ?, ?)";

    getSimpleJdbcTemplate().update(sql, vehicle.getVehicleNo(),
      vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat());
```


Kup ksi k Pole ksi k

```
  }
  ...
}
```

Klasa SimpleJdbcTemplate udostępnia wygodną metodę do masowego aktualizowania bazy. Pozwala ona podać instrukcję SQL-a i zestaw parametrów w kolekcji List<Object[]>, dzięki czemu nie trzeba implementować interfejsu BatchPreparedStatementSetter. Zauważ, że klasa SimpleJdbcTemplate wymaga albo obiektu typu DataSource, albo obiektu typu JdbcTemplate.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements VehicleDao {
  ...
  public void insertBatch(List<Vehicle> vehicles) {
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
             + "VALUES (?, ?, ?, ?)";

    List<Object[]> parameters = new ArrayList<Object[]>();
    for (Vehicle vehicle : vehicles) {
      parameters.add(new Object[] { vehicle.getVehicleNo(),
        vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
    }
    getSimpleJdbcTemplate().batchUpdate(sql, parameters);
  }
}
```

Używanie klasy SimpleJdbcTemplate do pobierania danych z bazy

Gdy implementujesz interfejs z rodziny RowMapper<T>, typem wartości zwracanej przez metodę mapRow() jest parametr z określonego interfejsu pochodnego. Tu jest to interfejs java.lang.Object.Parameterized ↪RowMapper<T>, przyjmujący jako parametr typ wartości zwracanej przez tę metodę.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.ParameterizedRowMapper;

public class VehicleRowMapper implements ParameterizedRowMapper<Vehicle> {
  public Vehicle mapRow(ResultSet rs, int rowNum) throws SQLException {
    Vehicle vehicle = new Vehicle();
    vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
    vehicle.setColor(rs.getString("COLOR"));
    vehicle.setWheel(rs.getInt("WHEEL"));
    vehicle.setSeat(rs.getInt("SEAT"));
    return vehicle;
  }
}
```

Zastosowanie klasy SimpleJdbcTemplate razem z interfejsem ParameterizedRowMapper<T> pozwala uniknąć rzutowania typu zwracanych wyników. W metodzie queryForObject() typ zwracanej wartości jest ustalany na podstawie parametru z obiektu ParameterizedRowMapper<T>. Tu jest to typ Vehicle. Zauważ, że parametry instrukcji trzeba podać na końcu listy argumentów, ponieważ ma ona zmienną długość.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
  VehicleDao {
```


Kup ksi k   Pole ksi k

```
  ...
  public Vehicle findByVehicleNo(String vehicleNo) {
    String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";

    // Nie trzeba już rzutować wyniku na typ Vehicle
    Vehicle vehicle = getSimpleJdbcTemplate().queryForObject(sql,
      new VehicleRowMapper(), vehicleNo);
    return vehicle;
  }
}
```

Spring udostępnia też wygodną implementację interfejsu `ParameterizedRowMapper<T>`. Jest to klasa `ParameterizedBeanPropertyRowMapper<T>`, która potrafi automatycznie odwzorować wiersz na nowy obiekt danej klasy.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.ParameterizedBeanPropertyRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
  VehicleDao {
  ...
  public Vehicle findByVehicleNo(String vehicleNo) {
    String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
    Vehicle vehicle = getSimpleJdbcTemplate().queryForObject(sql,
      ParameterizedBeanPropertyRowMapper.newInstance(Vehicle.class),
      vehicleNo);
    return vehicle;
  }
}
```

Gdy stosujesz klasyczną klasę `JdbcTemplate`, metoda `findAll()` powoduje wyświetlenie ostrzeżenia przez kompilator Javy, ponieważ występuje w niej niesprawdzana konwersja z typu `List` na `List<Vehicle>`. Jest tak, ponieważ typ wartości zwracanej przez metodę `query()` to `List`, a nie (bezpieczny ze względu na typ) `List<Vehicle>`. Po zastosowaniu klas `SimpleJdbcTemplate` i `ParameterizedBeanPropertyRowMapper<T>` to ostrzeżenie natychmiast znika, ponieważ zwracana kolekcja `List` zawiera obiekty tego samego typu co argument w `ParameterizedRowMapper<T>`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.ParameterizedBeanPropertyRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
  VehicleDao {
  ...
  public List<Vehicle> findAll() {
    String sql = "SELECT * FROM VEHICLE";

    List<Vehicle> vehicles = getSimpleJdbcTemplate().query(sql,
      ParameterizedBeanPropertyRowMapper.newInstance(Vehicle.class));
    return vehicles;
  }
}
```

Przy pobieraniu jednej wartości za pomocą klasy `SimpleJdbcTemplate` typ wartości zwracanej przez metodę `queryForObject()` jest określany na podstawie argumentu `class` (na przykład `String.class`). Nie trzeba więc ręcznie rzutować typu. Zauważ, że nieokreśloną liczbę dodatkowych parametrów instrukcji trzeba podać na końcu całej listy.

Kup ksi k Pole ksi k

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
  VehicleDao {
  ...
  public String getColor(String vehicleNo) {
    String sql = "SELECT COLOR FROM VEHICLE WHERE VEHICLE_NO = ?";

    // Nie trzeba już rzutować wartości na typ String
    String color = getSimpleJdbcTemplate().queryForObject(sql,
      String.class, vehicleNo);
    return color;
  }
}
```

15.5. Stosowanie nazwanych parametrów w szablonach JDBC

Problem

Przy klasycznym stosowaniu JDBC parametry SQL-a są reprezentowane przez symbol zastępczy ? i wiązane na podstawie pozycji. Problem z parametrami podawanymi za pomocą pozycji polega na tym, że po zmianie kolejności parametrów trzeba przestawić także wiązane z nimi wartości. W instrukcjach SQL-a o wielu parametrach dopasowywanie parametrów na podstawie pozycji jest bardzo kłopotliwe.

Rozwiązanie

Innym sposobem wiązania parametrów SQL-a w platformie Spring JDBC jest zastosowanie parametrów nazwanych. Jak wskazuje na to nazwa, takie parametry są określane na podstawie nazwy (poprzedzonej dwukropkiem), a nie według pozycji. Parametry nazwane ułatwiają zarządzanie kodem i poprawiają jego czytelność. W czasie wykonywania programu klasy platformy zastępują parametry nazwane symbolami zastępczymi. Parametry nazwane są obsługiwane tylko w klasach `SimpleJdbcTemplate` i `NamedParameterJdbcTemplate`.

Jak to działa?

Przy stosowaniu parametrów nazwanych w instrukcjach SQL-a możesz podać wartości parametrów w odwzorowaniu, w którym kluczami są nazwy parametrów.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
  VehicleDao {

  public void insert(Vehicle vehicle) {
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
           + "VALUES (:vehicleNo, :color, :wheel, :seat)";
```


Kup ksi k   Pole ksi k

```
    Map<String, Object> parameters = new HashMap<String, Object>();
    parameters.put("vehicleNo", vehicle.getVehicleNo());
    parameters.put("color", vehicle.getColor());
    parameters.put("wheel", vehicle.getWheel());
    parameters.put("seat", vehicle.getSeat());

    getSimpleJdbcTemplate().update(sql, parameters);
  }
  ...
}
```

Możesz też ustawić źródło parametrów SQL-a, odpowiedzialne za udostępnianie wartości nazwanych parametrów SQL-a. Istnieją trzy implementacje interfejsu `SqlParameterSource`. Podstawowym z nich jest `MapSqlParameterSource`, w którym źródłem parametrów jest odwzorowanie. Przedstawiony tu przykład jest mniej wydajny od poprzedniej wersji kodu, ponieważ wymaga dodatkowego obiektu (typu `SqlParameterSource`).

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
  VehicleDao {

  public void insert(Vehicle vehicle) {
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
               + "VALUES (:vehicleNo, :color, :wheel, :seat)";

    Map<String, Object> parameters = new HashMap<String, Object>();
    ...
    SqlParameterSource parameterSource =
      new MapSqlParameterSource(parameters);

    getSimpleJdbcTemplate().update(sql, parameterSource);
  }
  ...
}
```

Zalety tej techniki stają się widoczne wtedy, gdy potrzebny jest dodatkowy poziom pośredni między parametrami przekazywanymi do metody aktualizującej bazę a źródłem wartości. Jak na przykład pobrać właściwości z ziarna JavaBeans? Przydatny będzie do tego pośredni obiekt typu `SqlParameterSource`. Tu ten obiekt jest tworzony za pomocą typu `BeanPropertySqlParameterSource`, w którym zwykły obiekt Javy pełni funkcję źródła parametrów SQL-a. Dla każdego parametru nazwanego ustawiana jest wartość właściwości o tej samej nazwie.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements
  VehicleDao {

  public void insert(Vehicle vehicle) {
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
               + "VALUES (:vehicleNo, :color, :wheel, :seat)";
```


Kup ksi k Pole ksi k

```
    SqlParameterSource parameterSource =
      new BeanPropertySqlParameterSource(vehicle);

    getSimpleJdbcTemplate().update(sql, parameterSource);
  }
  ...
}
```

Parametry nazwane umożliwiają też masowe aktualizowanie bazy. Wartości parametrów można wtedy podać w tablicy z obiektami typu `Map` lub `SqlParameterSource`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport implements VehicleDao {
  ...
  public void insertBatch(List<Vehicle> vehicles) {
    String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
      + "VALUES (:vehicleNo, :color, :wheel, :seat)";

    List<SqlParameterSource> parameters = new ArrayList<SqlParameterSource>();
    for (Vehicle vehicle : vehicles) {
      parameters.add(new BeanPropertySqlParameterSource(vehicle));
    }

    getSimpleJdbcTemplate().batchUpdate(sql,
      parameters.toArray(new SqlParameterSource[0]));
  }
}
```

15.6. Obsługa wyjątków w platformie Spring JDBC

Problem

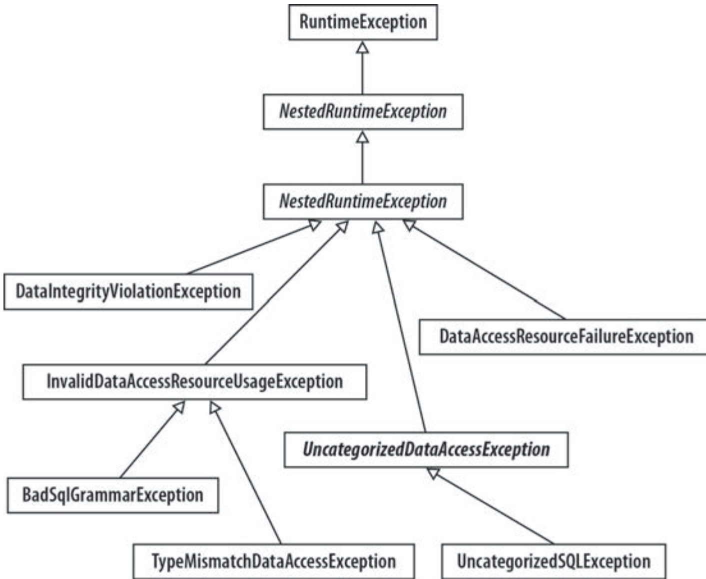
W wielu interfejsach API w JDBC zadeklarowane jest zgłaszanie wyjątków `java.sql.SQLException`. Są to wyjątki sprawdzane, które trzeba przechwycić. Bardzo kłopotliwe jest przechwytywanie takich wyjątków przy wykonywaniu każdej operacji na bazie danych. Często trzeba zdefiniować własną politykę ich obsługi. Jeśli tego nie zrobisz, obsługa wyjątków może być niespójna.

Rozwiązanie

Spring oferuje mechanizm spójnej obsługi wyjątków z zakresu dostępu do danych. Współdziała on z modułem dostępu do danych, w tym dla platformy JDBC. Wszystkie wyjątki zgłaszane przez platformę Spring JDBC są pochodne od klasy `org.springframework.dao.DataAccessException` (pochodnej od `RuntimeException`), a wyjątków tego typu nie trzeba przechwytywać. Jest to klasa bazowa dla wszystkich wyjątków w module dostępu do danych w Springu.

Rysunek 15.1 przedstawia tylko wycinek hierarchii wyjątków `DataAccessException` w module dostępu do danych Springa. W sumie istnieje ponad 30 klas przeznaczonych dla różnych kategorii wyjątków związanych z dostępem do danych.

Kup ksi k   Pole ksi k



Rysunek 15.1. *Często używane klasy wyjątków w hierarchii klasy DataAccessException*

Jak to działa?

Obsługa wyjątków w platformie Spring JDBC

Do tej pory nie obsługiwaliśmy bezpośrednio wyjątków JDBC przy stosowaniu szablonów JDBC lub obiektów do wykonywania operacji JDBC. Aby lepiej zrozumieć mechanizmy obsługi wyjątków platformy Spring JDBC, przyjrzyj się poniższemu fragmentowi kodu z klasy `Main` (ten fragment odpowiada za wstawianie pojazdu do bazy). Co się stanie, gdy wstawiony zostanie pojazd o numerze, który już znajduje się w bazie?

```
package com.apress.springrecipes.vehicle;
...
public class Main {

  public static void main(String[] args) {
    ...
    VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
    Vehicle vehicle = new Vehicle("EX0001", "Zielony", 4, 4);
    vehicleDao.insert(vehicle);
  }
}
```

Jeśli dwukrotnie uruchomisz tę metodę lub dany pojazd został już wstawiony do bazy, zgłoszony zostanie wyjątek typu `DuplicateKeyException` (ta klasa pośrednio dziedziczy po klasie `DataAccessException`). W metodach DAO nie trzeba umieszczać kodu w bloku `try-catch` ani deklarować w sygnaturze metody zgłaszania wyjątku. Wynika to z tego, że `DataAccessException` (a tym samym klasy od niej pochodne, w tym `DuplicateKeyException`) to klasa niesprawdzanego wyjątku, którego nie trzeba przechwytywać. Bezpośrednią klasą bazową dla `DataAccessException` jest `NestedRuntimeException` — podstawowa klasa wyjątków Springa, która umieszcza inny wyjątek w obiekcie typu `RuntimeException`.

Gdy używasz klas platformy Spring JDBC, przechwytują one automatycznie wyjątki `SQLException` i umieszczają je w obiektach klas pochodnych od `DataAccessException`. Ponieważ otrzymywany jest wtedy wyjątek `RuntimeException`, nie trzeba go przechwytywać.

Kup ksi k Pole ksi k

Skąd jednak platforma Spring JDBC ma „wiedzieć”, którą klasę z hierarchii klasy `DataAccessException` ma wybrać, aby zgłosić wyjątek? W tym celu sprawdzane są właściwości `errorCode` i `SQLState` przechwyconego wyjątku `SQLException`. Ponieważ klasa `DataAccessException` zapisuje dany wyjątek `SQLException` jako główną przyczynę problemu, wartości właściwości `errorCode` i `SQLState` można pobrać za pomocą poniższego bloku `catch`:

```
package com.apress.springrecipes.vehicle;
...
import java.sql.SQLException;

import org.springframework.dao.DataAccessException;

public class Main {

  public static void main(String[] args) {
    ...
    VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
    Vehicle vehicle = new Vehicle("EX0001", "Zielony", 4, 4);
    try {
      vehicleDao.insert(vehicle);
    } catch (DataAccessException e) {
      SQLException sqle = (SQLException) e.getCause();
      System.out.println("Kod błędu: " + sqle.getErrorCode());
      System.out.println("Stan SQL-a: " + sqle.getSQLState());
    }
  }
}
```

Gdy teraz spróbujesz wstawić pojazd o numerze istniejącym już w bazie, Apache Derby zwróci następujący kod błędu i stan SQL-a:

```
Kod błędu: -1
Stan SQL-a: 23505
```

W podręczniku użytkownika bazy Apache Derby znajdziesz opis tego kodu błędu (zobacz tabelę 15.2).

Tabela 15.2. *Opis kodu błędu z bazy Apache Derby*

Stan SQL-a	Tekst komunikatu
23505	Instrukcja została anulowana, ponieważ spowodowałaby wystąpienie powtarzającej się wartości w unikatowym polu, kluczu głównym lub unikatowym indeksie o identyfikatorze <*wartość*> zdefiniowanym w <*wartość*>.

Skąd platforma Spring JDBC „wie”, że stanowi `23505` odpowiada wyjątek typu `DuplicateKeyException`? Kody błędu i stany SQL-a są specyficzne dla baz danych, co oznacza, że w różnych bazach ten sam błąd może powodować zwrócenie odmiennych kodów. Ponadto niektóre bazy określają błąd za pomocą właściwości `errorCode`, natomiast inne (na przykład Derby) — przy użyciu właściwości `SQLState`.

Spring to otwarta platforma do tworzenia aplikacji w Javie, dlatego poprawnie interpretuje kody błędów z większości baz danych. Jednak ponieważ takich kodów jest dużo, uwzględniane są tylko najczęściej występujące problemy. Odwzorowanie kodów błędów jest zdefiniowane w pliku *sql-error-codes.xml* z pakietu `org.springframework.jdbc.support`. Poniższy fragment kodu dla bazy Apache Derby pochodzi właśnie z tego pliku.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 3.0//EN"
  "http://www.springframework.org/dtd/spring-beans-3.0.dtd">

<beans>
  ...
```


Kup ksi k   Pole ksi k

```
<bean id="Derby" class="org.springframework.jdbc.support.SQLErrorCodes">
  <property name="databaseProductName">
    <value>Apache Derby</value>
  </property>
  <property name="useSqlStateForTranslation">
    <value>true</value>
  </property>
  <property name="badSqlGrammarCodes">
    <value>42802,42821,42X01,42X02,42X03,42X04,42X05,42X06,
      ↪42X07,42X08</value>
    </property>
    <property name="duplicateKeyCodes">
      <value>23505</value>
    </property>
    <property name="dataIntegrityViolationCodes">
      <value>22001,22005,23502,23503,23513,
        ↪X0Y32</value>
    </property>
    <property name="dataAccessResourceFailureCodes">
      <value>04501,08004,42Y07</value>
    </property>
    <property name="cannotAcquireLockCodes">
      <value>40XL1</value>
    </property>
    <property name="deadlockLoserCodes">
      <value>40001</value>
    </property>
  </bean>
</beans>
```

Zauważ, że właściwość `databaseProductName` służy do dopasowywania nazwy bazy zwróconej przez metodę `Connection.getMetaData().getDatabaseProductName()`. Dzięki temu Spring potrafi ustalić, z jaką bazą się łączy. Właściwość `useSqlStateForTranslation` oznacza, że przy dopasowywaniu kodu błędu należy użyć właściwości `SQLState` zamiast właściwości `errorCode`. W klasie `SQLErrorCodes` zdefiniowanych jest kilka kategorii kodów błędów. Kod `23505` znajduje się w kategorii `dataIntegrityViolationCodes`.

Modyfikowanie obsługi wyjątków z obszaru dostępu do danych

Platforma Spring JDBC odwzorowuje tylko znane kody błędów. Czasem programista chce zmodyfikować odwzorowania. Może na przykład zdecydować się na dodanie nowych kodów do istniejących kategorii lub zdefiniowanie niestandardowego wyjątku dla konkretnych kodów błędu.

W opisie z tabeli 15.2 kod błędu `23505` oznacza powtarzający się klucz w bazie Apache Derby. Domyślnie ten kod jest odwzorowywany na wyjątek typu `DataIntegrityViolationException`. Załóżmy, że chcesz utworzyć dla błędów tego rodzaju niestandardowy typ wyjątków — `MyDuplicateKeyException`. Ten typ powinien dziedziczyć po klasie `DataIntegrityViolationException`, ponieważ też dotyczy błędu naruszenia integralności danych. Pamiętaj, że jeśli chcesz, aby wyjątek był zgłaszany przez platformę Spring JDBC, musi on być zgodny z główną klasą wyjątków `DataAccessException`.

```
package com.apress.springrecipes.vehicle;

import org.springframework.dao.DataIntegrityViolationException;

public class MyDuplicateKeyException extends DataIntegrityViolationException {

  public MyDuplicateKeyException(String msg) {
    super(msg);
  }
```


Kup ksi k   Pole ksi k

```
  public MyDuplicateKeyException(String msg, Throwable cause) {
    super(msg, cause);
  }
}
```

Domyślnie Spring szuka wyjątku w pliku *sql-error-codes.xml* z pakietu `org.springframework.jdbc.support`. Jeśli jednak chcesz przesłonić wybrane odwzorowania, możesz to zrobić za pomocą pliku o tej samej nazwie z katalogu głównego ustawionego w parametrze `classpath`. Po wykryciu takiego pliku Spring najpierw szuka wyjątku właśnie w nim. Gdy szukany wyjątek jest tu niedostępny, Spring szuka go w pliku domyślnym.

Załóżmy, że chcesz powiązać z kodem błędu 23505 niestandardowy typ `DuplicateKeyException`. Wymaga to dodania wiązania za pomocą ziarna `CustomSQLErrorCodesTranslation`, które to ziarno należy następnie dodać do kategorii `customTranslations`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
  <bean id="Derby"
    class="org.springframework.jdbc.support.SQLErrorCodes">
    <property name="databaseProductName">
      <value>Apache Derby</value>
    </property>
    <property name="useSqlStateForTranslation">
      <value>true</value>
    </property>
    <property name="customTranslations">
      <list>
        <ref local="myDuplicateKeyTranslation" />
      </list>
    </property>
  </bean>

  <bean id="myDuplicateKeyTranslation"
    class="org.springframework.jdbc.support.CustomSQLErrorCodesTranslation">
    <property name="errorCodes">
      <value>23505</value>
    </property>
    <property name="exceptionClass">
      <value>
        com.apress.springrecipes.vehicle.MyDuplicateKeyException
      </value>
    </property>
  </bean>
</beans>
```

Jeśli teraz usuniesz blok `try-catch` umieszczony wokół operacji wstawiania pojazdu i dodasz samochód o powtarzającym się numerze, platforma Spring JDBC zgłosi wyjątek `MyDuplicateKeyException`.

Jeżeli nie odpowiada Ci podstawowa strategia odwzorowywania kodów na wyjątki stosowana w klasie `SQLErrorCodes`, możesz zaimplementować interfejs `SQLExceptionTranslator` i wstrzykiwać obiekty tego typu do szablonu JDBC za pomocą metody `setExceptionTranslator()`.

Kup ksi k Pole ksi k

15.7. Problemy z bezpośrednim używaniem platform do tworzenia odwzorowań ORM

Problem

Programista zdecydował się przejść na następny poziom. Utworzył skomplikowany model domeny, dlatego ręczne pisanie kodu dla wszystkich encji jest żmudne. Programista rozpoczyna więc analizowanie różnych rozwiązań, na przykład Hibernate. Jest zaskoczony tym, że choć dają one duże możliwości, nie są proste w użyciu.

Rozwiązanie

Można wykorzystać Spring do pomocy. Spring udostępnia mechanizmy zarządzania warstwami odwzorowań ORM. Używanie tych mechanizmów jest tak proste jak dostęp do danych za pomocą standardowego interfejsu JDBC.

Jak to działa?

Załóżmy, że tworzysz system zarządzania kursami dla centrum szkoleniowego. Pierwszą klasą w tym systemie jest klasa `Course`. Jest to *klasa encji* (lub *klasa trwała*), ponieważ reprezentuje encję z rzeczywistego świata, a jej obiekty są utrwalane w bazie danych. Pamiętaj, że dla każdej klasy encji utrwalanej za pomocą platformy do tworzenia odwzorowań ORM potrzebny jest domyślny konstruktor bezargumentowy.

```
package com.apress.springrecipes.course;
...
public class Course {

  private Long id;
  private String title;
  private Date beginDate;
  private Date endDate;
  private int fee;

  // Konstruktory, gettery i settery
  ...
}
```

Dla każdej klasy encji trzeba zdefiniować właściwość pozwalającą na identyfikowanie obiektów tej klasy. Najlepszą praktyką jest zdefiniowanie automatycznie generowanego identyfikatora, ponieważ nie ma on znaczenia biznesowego, dlatego nigdy nie będzie trzeba modyfikować uzyskanych w ten sposób wartości. Ponadto w platformie do tworzenia odwzorowań ORM taki identyfikator jest używany do ustalania stanu encji. Jeśli wartość identyfikatora to `null`, encja jest traktowana jako nowa i niezapisana. Wtedy w momencie utrwalania encji zgłaszana jest instrukcja `insert` SQL-a. W przeciwnym razie korzysta się z instrukcji `update`. Aby identyfikator przyjmował wartości `null`, należy zastosować dla niego prosty typ nakładkowy (na przykład `java.lang.Integer` lub `java.lang.Long`).

W systemie zarządzania kursami potrzebny jest interfejs DAO zapewniający dostęp do danych. Zdefiniuj w interfejsie `CourseDao` następujące operacje:

```
package com.apress.springrecipes.course;
...
public interface CourseDao {

  public void store(Course course);
  public void delete(Long courseId);
```

```
  public Course findById(Long courseId);
  public List<Course> findAll();
}
```

Przy utrwalaniu obiektów za pomocą odwzorowań ORM zwykle operacje wstawiania i aktualizowania są łączone w jedno zadanie (zapisywanie). Dzięki temu to platforma do tworzenia odwzorowań ORM (a nie programista) decyduje, czy obiekt należy wstawić, czy zaktualizować.

Aby taka platforma mogła utrwalać obiekty w bazie danych, musi znać metadane dotyczące klas encji. Do platformy trzeba przekazać te metadane w odpowiednim formacie. W platformie Hibernate jest to format XML. Jednak ponieważ w poszczególnych platformach metadane mogą być definiowane w różnych formatach, w JPA zdefiniowany jest zestaw adnotacji związanych z utrwalaniem danych. Te adnotacje umożliwiają zapisywanie metadanych w standardowym formacie, dzięki czemu definicje łatwiej jest ponownie wykorzystać w innych platformach.

Hibernate także obsługuje adnotacje JPA służące do definiowania metadanych. Dlatego istnieją trzy różne strategie odwzorowywania i utrwalania obiektów za pomocą technologii Hibernate i JPA. Oto one:

- Wykorzystanie interfejsu API platformy Hibernate do utrwalania obiektów przy użyciu odwzorowań tej platformy (w formacie XML).
- Wykorzystanie interfejsu API platformy Hibernate do utrwalania obiektów za pomocą adnotacji JPA.
- Używanie JPA do utrwalania obiektów z wykorzystaniem adnotacji JPA.

Podstawowe elementy programistyczne w Hibernate, JPA i innych platformach do tworzenia odwzorowań ORM są podobne do rozwiązań z JDBC. Przegląd tych elementów przedstawia tabela 15.3.

Tabela 15.3. *Podstawowe elementy programistyczne powiązane z różnymi strategiami dostępu do danych*

Element	JDBC	Hibernate	JPA
Zasób	`Connection`	`Session`	`EntityManager`
Fabryka zasobów	`DataSource`	`SessionFactory`	`EntityManagerFactory`
Wyjątek	`SQLException`	`HibernateException`	`PersistenceException`

W Hibernate podstawowym interfejsem służącym do utrwalania obiektów jest `Session`. Obiekty tego typu można pobrać z obiektu typu `SessionFactory`. W JPA analogicznym interfejsem jest `EntityManager`, a obiekty tego typu są pobierane za pomocą obiektu typu `EntityManagerFactory`. Wyjątki zgłaszane przez `Hibernate` są typu `HibernateException`, natomiast wyjątki zgłaszane przez JPA mogą być typu `PersistenceException` lub mieć inny typ Javy SE (na przykład `IllegalArgumentException` lub `IllegalStateException`). Zauważ, że wszystkie te wyjątki dziedziczą po klasie `RuntimeException`, dlatego nie trzeba ich przechwytywać i obsługiwać.

Utrwalanie obiektów za pomocą interfejsu API platformy Hibernate i odwzorowań w formacie XML

Aby odwzorować klasy encji przy użyciu specyficznych dla platformy Hibernate odwzorowań w formacie XML, można utworzyć po jednym pliku dla każdej klasy albo duży plik przeznaczony dla kilku klas. W praktyce warto zdefiniować dla każdej klasy po jednym pliku o rozszerzeniu *.hbm.xml* (ułatwia to zarządzanie tymi plikami). Człon *hbm* to skrót od *Hibernate metadata*.

Plik z odwzorowaniem dla klasy `Course` powinien mieć nazwę *Course.hbm.xml* i znajdować się w tym samym pakiecie co klasa encji.

```
<!DOCTYPE hibernate-mapping
  PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.apress.springrecipes.course">
  <class name="Course" table="COURSE">
    <id name="id" type="long" column="ID">
      <generator class="identity" />
```


Kup ksi k   Pole ksi k

```
    </id>
    <property name="title" type="string">
      <column name="TITLE" length="100" not-null="true" />
    </property>
    <property name="beginDate" type="date" column="BEGIN_DATE" />
    <property name="endDate" type="date" column="END_DATE" />
    <property name="fee" type="int" column="FEE" />
  </class>
</hibernate-mapping>
```

W pliku z odwzorowaniem można podać nazwę tabeli odpowiadającej klasie encji i kolumny odpowiadające każdej właściwości typu prostego. Można też określić szczegółowe informacje na temat kolumn, na przykład długość kolumny, ograniczenia związane z wartością `null` i ograniczenia niepowtarzalności. Ponadto dla każdej encji trzeba zdefiniować identyfikator. Można go generować automatycznie lub przypisywać ręcznie. W tym przykładzie identyfikator jest generowany automatycznie.

Każda aplikacja używająca platformy Hibernate wymaga globalnego pliku z konfiguracją. Należy w nim skonfigurować różne właściwości: ustawienia bazy danych (albo właściwości połączenia JDBC, albo nazwę JNDI źródła danych), dialekt używany w danej bazie, lokalizację metadanych itd. Jeśli metadane są zdefiniowane w plikach odwzorowań w formacie XML, trzeba wskazać lokalizację tych plików. Domyślnie Hibernate wczytuje plik *hibernate.cfg.xml* z katalogu głównego ustawionego w parametrze `classpath`. Człon *cfg* to skrót od *configuration*. Jeśli w ścieżce z parametru `classpath` dostępny jest plik *hibernate.properties*, plik ten jest sprawdzany jako pierwszy, po czym ustawienia zostają zastąpione wartościami z pliku *hibernate.cfg.xml*.

```
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">
      org.apache.derby.jdbc.ClientDriver
    </property>

    <property name="connection.url">
      jdbc:derby://localhost:1527/course;create=true
    </property>
    <property name="connection.username">app</property>
    <property name="connection.password">app</property>
    <property name="dialect">org.hibernate.dialect.DerbyDialect</property>
    <property name="show_sql">true</property>
    <property name="hbm2ddl.auto">update</property>

    <mapping resource="com/apress/springrecipes/course/
      ➥Course.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

Zanim będzie można utrwalić obiekty, należy utworzyć w schemacie bazy tabele na dane tych obiektów. Gdy stosuje się platformę do tworzenia odwzorowań ORM, na przykład Hibernate, zwykle nie trzeba samemu projektować tabel. Ustawienie właściwości `hbm2ddl.auto` na wartość `update` powoduje, że Hibernate będzie pomagać w zakresie aktualizowania bazy i w razie potrzeby tworzyć tabele. Oczywiście nie należy stosować tej techniki w produkcyjnej wersji aplikacji, jednak można w ten sposób znacznie przyspieszyć tworzenie kodu.

Zaimplementujmy teraz w podpakiecie `hibernate` interfejs DAO za pomocą samego interfejsu API Hibernate. Zanim zaczniesz utrwalać obiekty przy użyciu tego interfejsu, musisz w konstruktorze zainicjować fabrykę sesji platformy Hibernate.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.Query;
```


Kup książkę   Poleć książkę

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class HibernateCourseDao implements CourseDao {

  private SessionFactory sessionFactory;

  public HibernateCourseDao() {
    Configuration configuration = new Configuration().configure();
    sessionFactory = configuration.buildSessionFactory();
  }

  public void store(Course course) {
    Session session = sessionFactory.openSession();
    Transaction tx = session.getTransaction();
    try {
      tx.begin();
      session.saveOrUpdate(course);
      tx.commit();
    } catch (RuntimeException e) {
      tx.rollback();
      throw e;
    } finally {
      session.close();
    }
  }

  public void delete(Long courseId) {
    Session session = sessionFactory.openSession();
    Transaction tx = session.getTransaction();
    try {
      tx.begin();
      Course course = (Course) session.get(Course.class, courseId);
      session.delete(course);
      tx.commit();
    } catch (RuntimeException e) {
      tx.rollback();
      throw e;
    } finally {
      session.close();
    }
  }

  public Course findById(Long courseId) {
    Session session = sessionFactory.openSession();
    try {
      return (Course) session.get(Course.class, courseId);
    } finally {
      session.close();
    }
  }

  public List<Course> findAll() {
    Session session = sessionFactory.openSession();
    try {
      Query query = session.createQuery("from Course");
```


Kup książkę Poleć książkę

```
      return query.list();
    } finally {
      session.close();
    }
  }
}
```

Pierwszy krok przy stosowaniu platformy Hibernate polega na utworzeniu obiektu typu `Configuration` i zażądaniu wczytania pliku konfiguracyjnego tej platformy. Wywołanie metody `configure()` domyślnie powoduje wczytanie pliku *hibernate.cfg.xml* z katalogu głównego ustawionego w parametrze `classpath`. Następnie za pomocą wspomnianego obiektu typu `Configuration` tworzona jest fabryka sesji platformy Hibernate. Fabryka sesji służy do generowania sesji na potrzeby utrwalania obiektów.

W przedstawionych wcześniej metodach DAO najpierw za pomocą fabryki sesji otwierana jest sesja. Każda operacja obejmująca aktualizowanie bazy (na przykład `saveOrUpdate()` i `delete()`) wymaga uruchomienia transakcji w tej sesji. Jeśli dana operacja zakończy się powodzeniem, transakcja jest zatwierdzana, natomiast wystąpienie wyjątków `RuntimeException` prowadzi do wycofania transakcji. W operacjach służących tylko do odczytu (takich jak `get()` lub zapytania w HQL-u) transakcje nie są potrzebne. Ponadto należy pamiętać o zamknięciu sesji w celu zwolnienia zajmowanych zasobów.

W celu testowego uruchomienia wszystkich metod DAO utwórzmy poniższą klasę `Main`. Ten kod ilustruje też typowy cykl życia encji.

```
package com.apress.springrecipes.course;
...
public class Main {

  public static void main(String[] args) {
    CourseDao courseDao = new HibernateCourseDao();

    Course course = new Course();
    course.setTitle("Podstawy Springa");
    course.setBeginDate(new GregorianCalendar(2007, 8, 1).getTime());
    course.setEndDate(new GregorianCalendar(2007, 9, 1).getTime());
    course.setFee(1000);
    courseDao.store(course);

    List<Course> courses = courseDao.findAll();
    Long courseId = courses.get(0).getId();

    course = courseDao.findById(courseId);
    System.out.println("Tytuł kursu: " + course.getTitle());
    System.out.println("Data rozpoczęcia: " + course.getBeginDate());
    System.out.println("Data zakończenia: " + course.getEndDate());
    System.out.println("Cena: " + course.getFee());

    courseDao.delete(courseId);
  }
}
```

Utrwalanie obiektów za pomocą interfejsu API Hibernate i adnotacji JPA

Adnotacje JPA są opisane w specyfikacji JSR-220, dlatego ich obsługa jest dostępna we wszystkich zgodnych z JPA platformach do tworzenia odwzorowań ORM (w tym w platformie Hibernate). Ponadto stosowanie adnotacji sprawia, że łatwiej jest modyfikować metadane odwzorowań zapisane w jednym pliku źródłowym.

Poniższa klasa `Course` pokazuje, jak zdefiniować metadane odwzorowań za pomocą adnotacji JPA.

```
package com.apress.springrecipes.course;
...
import javax.persistence.Column;
import javax.persistence.Entity;
```


Kup ksi k   Pole ksi k

```
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "COURSE")
public class Course {

  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  @Column(name = "ID")
  private Long id;

  @Column(name = "TITLE", length = 100, nullable = false)
  private String title;

  @Column(name = "BEGIN_DATE")
  private Date beginDate;

  @Column(name = "END_DATE")
  private Date endDate;

  @Column(name = "FEE")
  private int fee;

  // Konstruktory, gettery i settery
  ...
}
```

Każda klasa encji musi być opatrzona adnotacją `@Entity`. W tej adnotacji można przypisać do tej klasy nazwę tabeli. Dla każdej właściwości można za pomocą adnotacji `@Column` ustawić nazwę kolumny i informacje na jej temat. Każda klasa encji musi mieć identyfikator podany w adnotacji `@Id`. Za pomocą adnotacji `@GeneratedValue` można określić strategię generowania identyfikatorów. Tu identyfikatory są tworzone na podstawie kolumny z automatycznie generowanymi wartościami.

Hibernate umożliwia definiowanie metadanych odwzorowań zarówno za pomocą natywnych plików z odwzorowaniami w formacie XML, jak i adnotacji JPA. Przy stosowaniu adnotacji JPA trzeba podać pełne nazwy klas encji w pliku *hibernate.cfg.xml*, aby platforma Hibernate wczytała adnotacje.

```
<hibernate-configuration>
  <session-factory>
    ...
    <!-- Przy stosowaniu odwzorowań w formacie XML -->
    <!--
    <mapping resource="com/apress/springrecipes/course/
      ↪Course.hbm.xml" />
    -->

    <!-- Przy stosowaniu adnotacji JPA -->
    <mapping class="com.apress.springrecipes.course.Course" />
  </session-factory>
</hibernate-configuration>
```

W implementacji DAO używanej w platformie Hibernate klasa `Configuration` służy do wczytywania odwzorowań w formacie XML. Jeśli definiujesz metadane odwzorowań dla platformy Hibernate za pomocą adnotacji JPA, musisz zastosować klasę pochodną `AnnotationConfiguration`.

Kup ksi k Pole ksi k

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

public class HibernateCourseDao implements CourseDao {

  private SessionFactory sessionFactory;

  public HibernateCourseDao() {
    // Przy stosowaniu odwzorowań w formacie XML platformy Hibernate
    // Configuration configuration = new Configuration().configure();

    // Przy stosowaniu adnotacji JPA
    Configuration configuration = new AnnotationConfiguration().configure();
    sessionFactory = configuration.buildSessionFactory();
  }
  ...
}
```

Utrwalanie obiektów za pomocą JPA i implementacji w postaci platformy Hibernate

Oprócz dotyczących utrwalania adnotacji w JPA zdefiniowany jest zestaw interfejsów programowania umożliwiający utrwalanie obiektów. Jednak JPA nie jest implementacją operacji utrwalania danych. Aby zapewnić usługi z tego obszaru, trzeba zastosować silnik zgodny z JPA. Zgodność platformy Hibernate z JPA można zapewnić za pomocą rozszerzenia w postaci modułu Hibernate EntityManager. To sprawia, że Hibernate może działać jako silnik utrwalania obiektów w JPA. Dzięki temu można nadal korzystać z Hibernate (możliwe, że platforma ta jest szybsza i lepiej obsługuje niektóre operacje), a jednocześnie pisać kod zgodny z JPA i współdziałający także z innymi silnikami JPA. Takie podejście jest też przydatne do dostosowywania kodu do JPA. Nowy kod jest wtedy pisany pod kątem interfejsów API z JPA, a starszy kod należy zmodyfikować pod kątem JPA.

W Javie EE można skonfigurować silnik JPA w kontenerze Javy EE. W aplikacjach Javy SE silnik trzeba ustawić lokalnie. Konfigurowanie JPA odbywa się za pomocą centralnego pliku *persistence.xml* umieszczonego w katalogu *META-INF* w katalogu głównym określonym w parametrze `classpath`. W tym pliku można ustawić specyficzne dla producenta właściwości, aby skonfigurować używany silnik.

Teraz utwórzmy plik konfiguracyjny JPA *persistence.xml*. Należy go umieścić w katalogu *META-INF* w katalogu głównym ustawionym w parametrze `classpath`. Każdy plik konfiguracyjny JPA zawiera przynajmniej jeden element `<persistence-unit>`. Określa on *jednostkę utrwalania*, która definiuje zestaw utrwalanych klas i sposób ich utrwalania. Każda jednostka utrwalania musi mieć nazwę. Tu jest nią `course`.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
    http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
  version="1.0">

  <persistence-unit name="course">
    <properties>
      <property name="hibernate.ejb.cfgfile" value="/hibernate.cfg.xml" />
    </properties>
  </persistence-unit>
</persistence>
```

W tym pliku konfiguracyjnym JPA jako silnik JPA ustawiana jest platforma Hibernate. W tym celu należy wskazać plik konfiguracyjny Hibernate zapisany w katalogu głównym określonym w parametrze `classpath`. Ponieważ rozszerzenie Hibernate EntityManager automatycznie wykrywa metadane odwzorowań w postaci

Kup ksi k Pole ksi k

plików w formacie XML i adnotacji JPA, nie trzeba bezpośrednio ich ustawiać. W przeciwnym razie wystąpi wyjątek `org.hibernate.DuplicateMappingException`.

```
<hibernate-configuration>
  <session-factory>
    ...
    <!-- Nie trzeba ustawiać plików z odwzorowaniami i klas z adnotacjami -->
    <!--
      <mapping resource="com/apress/springrecipes/course/
        ➥Course.hbm.xml" />
      <mapping class="com.apress.springrecipes.course.Course" />
    -->
  </session-factory>
</hibernate-configuration>
```

Zamiast wskazywać plik konfiguracyjny Hibernate, można też scentralizować całą konfigurację Hibernate w pliku *persistence.xml*.

```
<persistence ...>
  <persistence-unit name="course">
    <properties>
      <property name="hibernate.connection.driver_class"
        value="org.apache.derby.jdbc.ClientDriver" />
      <property name="hibernate.connection.url"
        value="jdbc:derby://localhost:1527/course;create=true" />
      <property name="hibernate.connection.username" value="app" />
      <property name="hibernate.connection.password" value="app" />
      <property name="hibernate.dialect"
        value="org.hibernate.dialect.DerbyDialect" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.hbm2ddl.auto" value="update" />
    </properties>
  </persistence-unit>
</persistence>
```

W Javie EE kontener może zarządzać menedżerem encji za programistę i wstrzykiwać taki menedżer bezpośrednio do komponentów EJB. Jednak gdy używa się JPA poza kontenerem Javy EE (na przykład w aplikacji Javy SE), trzeba samodzielnie utworzyć menedżer encji i zarządzać nim.

■ **Uwaga** Aby wykorzystać Hibernate jako silnik JPA, trzeba ustawić w parametrze `classpath` biblioteki rozszerzenia Hibernate EntityManager. Jeśli używasz Mavena, dodaj do projektu poniższą zależność:

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

Teraz zaimplementujmy interfejs `CourseDao` w podpakiecie `jpa`, używając JPA w aplikacji Javy SE. Przed wywołaniem JPA w celu utrwalenia obiektów trzeba zainicjować fabrykę menedżerów encji. Fabryka ta ma generować menedżery encji służące do utrwalania obiektów.

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
```


Kup ksi k Pole ksi k

```
import javax.persistence.Query;

public class JpaCourseDao implements CourseDao {

  private EntityManagerFactory entityManagerFactory;

  public JpaCourseDao() {
    entityManagerFactory = Persistence.createEntityManagerFactory("course");
  }

  public void store(Course course) {
    EntityManager manager = entityManagerFactory.createEntityManager();
    EntityTransaction tx = manager.getTransaction();
    try {
      tx.begin();
      manager.merge(course);
      tx.commit();
    } catch (RuntimeException e) {
      tx.rollback();
      throw e;
    } finally {
      manager.close();
    }
  }

  public void delete(Long courseId) {
    EntityManager manager = entityManagerFactory.createEntityManager();
    EntityTransaction tx = manager.getTransaction();
    try {
      tx.begin();
      Course course = manager.find(Course.class, courseId);
      manager.remove(course);
      tx.commit();
    } catch (RuntimeException e) {
      tx.rollback();
      throw e;
    } finally {
      manager.close();
    }
  }

  public Course findById(Long courseId) {
    EntityManager manager = entityManagerFactory.createEntityManager();
    try {
      return manager.find(Course.class, courseId);
    } finally {
      manager.close();
    }
  }

  public List<Course> findAll() {
    EntityManager manager = entityManagerFactory.createEntityManager();
    try {
      Query query = manager.createQuery(
        "select course from Course course");
      return query.getResultList();
    } finally {
      manager.close();
```


Kup ksi   k   Pole   ksi   k

```
    }
  }
}
```

Fabrykę menedżerów encji można utworzyć za pomocą statycznej metody `createEntityManagerFactory()` klasy `javax.persistence.Persistence`. Należy przy tym przekazać nazwę jednostki utrwalania zdefiniowaną w pliku *persistence.xml*.

We wcześniejszych metodach DAO trzeba najpierw utworzyć menedżer encji za pomocą fabryki takich menedżerów. W operacjach modyfikujących bazę danych (takich jak `merge()` i `remove()`) należy przy użyciu menedżera encji rozpocząć transakcję JPA. W operacjach służących tylko do odczytu (takich jak `find()` i zapytania JPA) transakcje nie są potrzebne. Na zakończenie należy zamknąć menedżer encji, aby zwolnić zasoby.

Te metody DAO można przetestować za pomocą klasy `Main` podobnej do jej wcześniejszej wersji, jednak tym razem trzeba zastosować implementację DAO opartą na JPA.

```
package com.apress.springrecipes.course;
...
public class Main {

  public static void main(String[] args) {
    CourseDao courseDao = new JpaCourseDao();
    ...
  }
}
```

W przedstawionych implementacjach DAO opartych na Hibernate i JPA poszczególne metody DAO różnią się tylko jednym lub dwoma wierszami kodu. Pozostałe wiersze wykonują szablonowe, rutynowe zadania, które trzeba powtarzać. Ponadto każda platforma do tworzenia odwzorowań ORM ma własny interfejs API służący do zarządzania lokalnymi transakcjami.

15.8. Konfigurowanie fabryk zasobów ORM w Springu

Problem

Przy stosowaniu platformy do tworzenia odwzorowań ORM trzeba skonfigurować fabrykę zasobów za pomocą interfejsu API tej platformy. W Hibernate i JPA fabrykę sesji oraz fabrykę menedżerów encji można utworzyć przy użyciu natywnego interfejsu API Hibernate i JPA. Bez pomocy ze strony Springa nie da się utworzyć takich obiektów ręcznie.

Rozwiązanie

Spring udostępnia kilka ziaren fabrycznych, które umożliwiają utworzenie fabryki sesji Hibernate lub fabryki menedżerów encji JPA jako ziarna typu singleton w kontenerze IoC. Te fabryki można dzięki wstrzykiwaniu zależności współużytkować w różnych ziarnach. Ponadto fabrykę sesji i fabrykę menedżerów encji można zintegrować z działającymi w Springu mechanizmami dostępu do danych, takimi jak źródła danych i menedżery transakcji.

Kup ksi k Pole ksi k

Jak to działa?

Konfigurowanie w Springu fabryki sesji platformy Hibernate

Zacznijmy od zmodyfikowania klasy `HibernateCourseDao` w taki sposób, aby przyjmowała fabrykę sesji za pomocą wstrzykiwania zależności. Ta technika zastąpi tu bezpośrednie tworzenie fabryki sesji w konstruktorze przy użyciu natywnego interfejsu API platformy Hibernate.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;

public class HibernateCourseDao implements CourseDao {
  private SessionFactory sessionFactory;

  public void setSessionFactory(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
  }
  ...
}
```

Teraz przyjrzyj się temu, jak w Springu zadeklarować fabrykę sesji, która używa plików z odwzorowaniami w formacie XML. W tym celu trzeba ponownie ustawić definicję z takim plikiem w pliku *hibernate.cfg.xml*.

```
<hibernate-configuration>
  <session-factory>
    ...
    <!-- Dla odwzorowań Hibernate w formacie XML -->
    <mapping resource="com/apress/springrecipes/course/
      ↪Course.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

Następnie utwórzmy plik z konfiguracją ziaren, który pozwoli wykorzystać Hibernate jako platformę do tworzenia odwzorowań ORM. Ten plik należy nazwać *beans-hibernate.xml* i zapisać w katalogu głównym ustawionym w parametrze `classpath`. Za pomocą ziarna fabrycznego `LocalSessionFactoryBean` można tu zadeklarować fabrykę sesji, która używa pliku z odwzorowaniami w formacie XML. Ponadto można zadeklarować obiekt typu `HibernateCourseDao` zarządzany przez Spring.

```
<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="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="configLocation" value="classpath:hibernate.cfg.xml" />
  </bean>

  <bean id="courseDao"
    class="com.apress.springrecipes.course.hibernate.
      ↪HibernateCourseDao">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>
</beans>
```

Zauważ, że można ustawić właściwość `configLocation`, aby to ziarno fabryczne wczytywało plik konfiguracyjny platformy Hibernate. Właściwość `configLocation` jest typu `Resource`, jednak można przypisać do niej łańcuch znaków. Wbudowany edytor właściwości, `ResourceEditor`, przekształca taki łańcuch na obiekt

Kup ksi k Pole ksi k

typu Resource. Wcześniejsze ziarno fabryczne wczytuje plik konfiguracyjny z katalogu głównego ustawionego w parametrze classpath.

Teraz można zmodyfikować klasę Main, aby pobierała obiekt typu HibernateCourseDao z kontenera IoC.

```
package com.apress.springrecipes.course;
...
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

  public static void main(String[] args) {
    ApplicationContext context =
      new ClassPathXmlApplicationContext("beans-hibernate.xml");

    CourseDao courseDao = (CourseDao) context.getBean("courseDao");
    ...
  }
}
```

Przedstawione ziarno fabryczne tworzy fabrykę sesji w wyniku wczytania pliku konfiguracyjnego platformy Hibernate. Plik ten obejmuje ustawienia bazy danych (właściwości połączenia JDBC lub nazwę JNDI źródła danych). Załóżmy, że w kontenerze IoC zdefiniowane jest źródło danych. Jeśli chcesz wykorzystać to źródło danych w fabryce sesji, możesz wstrzyknąć je do właściwości dataSource ziarna LocalSessionFactoryBean. Źródło danych zapisane w tej właściwości zastępuje ustawienia bazy danych z pliku konfiguracyjnego platformy Hibernate. Jeśli używane jest źródło danych, w ustawieniach platformy Hibernate nie należy definiować dostawcy połączeń. Pozwala to uniknąć niepotrzebnej podwójnej konfiguracji.

```
<beans ...>
  ...
  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
      value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url"
      value="jdbc:derby://localhost:1527/course;create=true" />
    <property name="username" value="app" />
    <property name="password" value="app" />
  </bean>

  <bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="configLocation" value="classpath:hibernate.cfg.xml" />
  </bean>
</beans>
```

Możesz nawet zrezygnować z pliku konfiguracyjnego platformy Hibernate. W tym celu scal całą konfigurację w pliku LocalSessionFactoryBean. Na przykład możesz ustawić we właściwości mappingResources lokalizację plików z odwzorowaniami w formacie XML, a we właściwości hibernateProperties — określić dialekt bazy danych.

```
<bean id="sessionFactory"
  class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="mappingResources">
    <list>
      <value>com/apress/springrecipes/course/Course.hbm.xml</value>
    </list>
  </property>
```


Kup ksi k Pole ksi k

```
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
      <prop key="hibernate.show_sql">true</prop>
      <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
  </property>
</bean>
```

Właściwość `mappingResources` jest typu `String[]`, dlatego można ustawić w parametrze `classpath` zestaw plików z odwzorowaniami. Ziarno `LocalSessionFactoryBean` umożliwia też wykorzystanie dostępnego w Springu mechanizmu wczytywania zasobów. Ten mechanizm umożliwia wczytywanie plików z odwzorowaniami z różnych lokalizacji. We właściwości `mappingLocations` (typu `Resource[]`) można podać ścieżki do zasobów w postaci plików z odwzorowaniami.

```
<bean id="sessionFactory"
  class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  ...
  <property name="mappingLocations">
    <list>
      <value>classpath:com/apress/springrecipes/course/Course.hbm.xml</value>
    </list>
  </property>
  ...
</bean>
```

Przy stosowaniu dostępnego w Springu mechanizmu wczytywania zasobów można też podać w ścieżce symbole wieloznaczne, aby pasowała ona do różnych plików z odwzorowaniami. Dzięki temu nie trzeba ustawiać ich lokalizacji po dodaniu nowych klas encji. Wstępnie rejestrowany w Springu edytor `ResourceArrayProperty`
`↪Editor` przekształca podaną ścieżkę na tablicę obiektów typu `Resource`.

```
<bean id="sessionFactory"
  class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
  ...
  <property name="mappingLocations"
    value="classpath:com/apress/springrecipes/course/*.hbm.xml" />
  ...
</bean>
```

Jeśli metadane odwzorowania są podawane za pomocą adnotacji JPA, trzeba zastosować ziarno `AnnotationSessionFactoryBean`. Należy podać utrwalane klasy we właściwości `annotatedClasses` tego ziarna lub wykorzystać jego właściwość `packagesToScan`, aby poinformować, które pakiety należy przeskanować pod kątem adnotacji JPA.

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.
  ↪annotation.AnnotationSessionFactoryBean">
  <property name="dataSource" ref="dataSource" />
  <property name="annotatedClasses">
    <list>
      <value>com.apress.springrecipes.course.Course</value>
    </list>
  </property>
  <property name="hibernateProperties">
    <props>
      <prop key="hibernate.dialect">org.hibernate.dialect.DerbyDialect</prop>
      <prop key="hibernate.show_sql">true</prop>
      <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
  </property>
</bean>
```


Kup ksi k Pole ksi k

Teraz można usunąć plik konfiguracyjny platformy Hibernate (*hibernate.cfg.xml*), ponieważ ustawienia z tego pliku przeniesiono do Springa.

Konfigurowanie fabryki menedżerów encji JPA w Springu

Przede wszystkim trzeba zmodyfikować klasę `JpaCourseDao`, aby pobierała fabrykę menedżerów encji za pomocą wstrzykiwania zależności, zamiast generować ją bezpośrednio w konstruktorze.

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManagerFactory;
import javax.persistence.Persistence;

public class JpaCourseDao implements CourseDao {

  private EntityManagerFactory entityManagerFactory;

  public void setEntityManagerFactory(
    EntityManagerFactory entityManagerFactory) {
    this.entityManagerFactory = entityManagerFactory;
  }
  ...
}
```

Specyfikacja JPA definiuje, w jaki sposób należy pobierać fabrykę menedżerów encji w Javie SE i Javie EE. W Javie SE taką fabrykę tworzy się ręcznie w wyniku wywołania statycznej metody `createEntityManagerFactory()` klasy `Persistence`.

Utwórzmy plik z konfiguracją ziaren na potrzeby JPA. Powinien to być plik *beans-jpa.xml* w katalogu głównym ustawionym w parametrze `classpath`. Spring udostępnia ziarno fabryczne `LocalEntityManagerFactoryBean` przeznaczone do tworzenia fabryki menedżerów encji w kontenerze IoC. Programista musi określić nazwę jednostki utrwalania zdefiniowanej w pliku konfiguracyjnym JPA. Można też zadeklarować obiekt typu `JpaCourseDao` zarządzany przez Spring.

```
<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="entityManagerFactory"
    class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="course" />
  </bean>

  <bean id="courseDao"
    class="com.apress.springrecipes.course.jpa.JpaCourseDao">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
  </bean>
</beans>
```

Teraz możesz pobrać obiekt typu `JpaCourseDao` z kontenera IoC i przetestować ten obiekt w klasie `Main`.

```
package com.apress.springrecipes.course;
...
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

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


Kup ksi k Pole ksi k

```
      new ClassPathXmlApplicationContext("beans-jpa.xml");

    CourseDao courseDao = (CourseDao) context.getBean("courseDao");
    ...
  }
}
```

W Javie EE można pobrać fabrykę menedżerów encji z kontenera Javy EE na podstawie nazwy JNDI. W Springu, aby przeprowadzić wyszukiwanie nazw JNDI, należy zastosować element `<jee:jndi-lookup>`.

```
<jee:jndi-lookup id="entityManagerFactory" jndi-name="jpa/coursePU" />
```

Ziarno typu `LocalEntityManagerFactoryBean` tworzy fabrykę menedżerów encji w wyniku wczytania pliku konfiguracyjnego JPA (jest to plik *persistence.xml*). Spring obsługuje też bardziej elastyczny sposób tworzenia fabryki menedżerów encji. Służy do tego ziarno typu `LocalContainerEntityManagerFactoryBean`. Umożliwia ono przesłonienie wybranych ustawień (na przykład źródła danych i dialektu używanego w bazie) w pliku konfiguracyjnym JPA. Można więc wykorzystać oferowane przez Spring mechanizmy dostępu do danych, aby skonfigurować fabrykę menedżerów encji.

```
<beans ...>
  ...
  <bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
      value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url"
      value="jdbc:derby://localhost:1527/course;create=true" />
    <property name="username" value="app" />
    <property name="password" value="app" />
  </bean>

  <bean id="entityManagerFactory" class="org.springframework.orm.jpa.
    LocalContainerEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="course" />
    <property name="dataSource" ref="dataSource" />
    <property name="jpaVendorAdapter">
      <bean class="org.springframework.orm.jpa.vendor.
        ↪HibernateJpaVendorAdapter">
        <property name="databasePlatform"
          value="org.hibernate.dialect.DerbyDialect" />
        <property name="showSql" value="true" />
        <property name="generateDdl" value="true" />
      </bean>
    </property>
  </bean>
</beans>
```

W tej konfiguracji źródło danych jest wstrzykiwane do fabryki menedżerów encji. To powoduje zastąpienie ustawień bazy zapisanych w pliku z konfiguracją JPA. Aby ustawić właściwości specyficzne dla silnika JPA, należy ustawić adapter implementacji JPA na wartość `LocalContainerEntityManagerFactoryBean`. Gdy jako silnika JPA używa się platformy Hibernate, należy zastosować adapter `HibernateJpaVendorAdapter`. Właściwości nieobsługiwane przez ten adapter można podać we właściwości `jpaProperties`.

Teraz plik konfiguracyjny JPA (plik *persistence.xml*) można uprościć w przedstawiony poniżej sposób, ponieważ zapisane w nim ustawienia przeniesiono do Springa.

```
<persistence ...>
  <persistence-unit name="course" />
</persistence>
```


Kup ksi k Pole ksi k

15.9. Utrwalanie obiektów za pomocą szablonów ORM Springa

Problem

Przy korzystaniu z platformy do tworzenia odwzorowań ORM trzeba wielokrotnie wykonywać te same rutynowe zadania dla każdej operacji DAO. Na przykład w operacji DAO implementowanej za pomocą Hibernate lub JPA należy otworzyć i zamknąć sesję lub menedżer encji, a także rozpocząć, zatwierdzić lub wycofać transakcję za pomocą natywnego interfejsu API.

Rozwiązanie

W Springu technika upraszczania korzystania z platformy do tworzenia odwzorowań ORM działa podobnie jak w JDBC — należy zdefiniować klasy szablonu i klasy DAO. Ponadto Spring definiuje warstwę abstrakcyjną na poziomie nad interfejsami API do zarządzania transakcjami. Dla różnych platform do tworzenia odwzorowań ORM wystarczy wybrać odpowiednią implementację menedżera transakcji. Następnie można zarządzać transakcjami w podobny sposób.

W module dostępu do danych w Springu obsługa różnych strategii takiego dostępu działa spójnie. W tabeli 15.4 wymienione są klasy współdziałające z technologiami JDBC, Hibernate i JPA.

Tabela 15.4. *Klasy Springa współdziałające z różnymi strategiami dostępu do danych*

Klasy	JDBC	Hibernate	JPA
Klasa szablonu	`JdbcTemplate`	`HibernateTemplate`	`JpaTemplate`
Obsługa DAO	`JdbcDaoSupport`	`HibernateDaoSupport`	`JpaDaoSupport`
Transakcja	`DataSourceTransaction`	`HibernateTransaction`	`JpaTransactionManager`

W Springu zdefiniowane są klasy `HibernateTemplate` i `JpaTemplate`, które zapewniają szablonowe metody dla różnych operacji związanych z Hibernate i JPA, co ułatwia korzystanie z tych technologii. Metody szablonowe z tych klas gwarantują, że sesje Hibernate i menedżery encji JPA będą poprawnie otwierane i zamykane. Ponadto klasy te sprawiają, że natywne transakcje Hibernate i JPA działają w ramach transakcji zarządzanych przez Spring. W efekcie programista może zarządzać transakcjami deklaratywnie w obiektach DAO związanych z Hibernate i JPA, przy czym nie wymaga to pisania powtarzającego się kodu do obsługi transakcji.

Jak to działa?

Stosowanie szablonów Hibernate i JPA

Klasę `HibernateCourseDao` można za pomocą klasy `HibernateTemplate` uprościć w przedstawiony poniżej sposób.

```
package com.apress.springrecipes.course.hibernate;
...
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.transaction.annotation.Transactional;

public class HibernateCourseDao implements CourseDao {

  private HibernateTemplate hibernateTemplate;

  public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
```


Kup ksi k Pole ksi k

```
    this.hibernateTemplate = hibernateTemplate;
  }

  @Transactional
  public void store(Course course) {
    hibernateTemplate.saveOrUpdate(course);
  }

  @Transactional
  public void delete(Long courseId) {
    Course course = (Course) hibernateTemplate.get(Course.class, courseId);
    hibernateTemplate.delete(course);
  }

  @Transactional(readOnly = true)
  public Course findById(Long courseId) {
    return (Course) hibernateTemplate.get(Course.class, courseId);
  }

  @Transactional(readOnly = true)
  public List<Course> findAll() {
    return hibernateTemplate.find("from Course");
  }
}
```

W tej implementacji interfejsu DAO wszystkie metody DAO są opatrzone adnotacją DAO, co oznacza, że mają działać w ramach transakcji. Spośród tych metod `findById()` i `findAll()` są przeznaczone tylko do odczytu. Metody szablonowe z klasy `HibernateTemplate` odpowiadają za zarządzanie sesjami i transakcjami. Jeśli w metodzie DAO działającej w ramach transakcji występuje wiele operacji platformy Hibernate, metody szablonowe gwarantują, że wszystkie te operacje zostaną wykonane w ramach tej samej sesji i transakcji. W efekcie do zarządzania sesjami i transakcjami nie trzeba używać interfejsu API tej platformy.

Klasa `HibernateTemplate` jest bezpieczna ze względu na wątki, dlatego można zadeklarować jeden obiekt tej klasy w pliku z konfiguracją ziaren dla platformy Hibernate (czyli w pliku *beans-hibernate.xml*) i wstrzykiwać ten obiekt do wszystkich obiektów DAO tej platformy. Dla obiektu typu `HibernateTemplate` trzeba ustawić właściwość `sessionFactory`. Tę właściwość można wstrzyknąć albo za pomocą settera, albo jako argument konstruktora.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
  ...
  <tx:annotation-driven />

  <bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>

  <bean id="hibernateTemplate"
    class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory" />
    </bean>

  <bean name="courseDao"
```

```
    class="com.apress.springrecipes.course.hibernate.
      ➥HibernateCourseDao">
    <property name="hibernateTemplate" ref="hibernateTemplate" />
  </bean>
</beans>
```

Aby włączyć deklaratywne zarządzanie transakcjami dla metod opatrzonych adnotacją `@Transactional`, trzeba ustawić element `<tx:annotation-driven>` w pliku z konfiguracją ziaren. Domyślnie używany jest menedżer transakcji o nazwie `transactionManager`, dlatego trzeba zadeklarować obiekt typu `HibernateTransaction➥Manager` o tej nazwie. Dla tego obiektu należy ustawić właściwość określającą fabrykę sesji. Ten obiekt zarządza transakcjami w sesjach otwartych za pomocą podanej fabryki.

W podobny sposób można wykorzystać klasę `JpaTemplate` Springa, aby uprościć klasę `JpaCourseDao`. Także tu należy zadeklarować wszystkie metody DAO jako metody transakcyjne.

```
package com.apress.springrecipes.course.jpa;
...
import org.springframework.orm.jpa.JpaTemplate;
import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao implements CourseDao {

  private JpaTemplate jpaTemplate;

  public void setJpaTemplate(JpaTemplate jpaTemplate) {
    this.jpaTemplate = jpaTemplate;
  }

  @Transactional
  public void store(Course course) {
    jpaTemplate.merge(course);
  }

  @Transactional
  public void delete(Long courseId) {
    Course course = jpaTemplate.find(Course.class, courseId);
    jpaTemplate.remove(course);
  }

  @Transactional(readOnly = true)
  public Course findById(Long courseId) {
    return jpaTemplate.find(Course.class, courseId);
  }

  @Transactional(readOnly = true)
  public List<Course> findAll() {
    return jpaTemplate.find("from Course");
  }
}
```

W pliku z konfiguracją ziaren dla JPA (czyli w pliku *beans-jpa.xml*) można zadeklarować obiekt typu `JpaTemplate`, a następnie wstrzykiwać go do wszystkich metod DAO związanych z JPA. Ponadto na potrzeby zarządzania transakcjami JPA trzeba zadeklarować obiekt typu `JpaTransactionManager`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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/tx
```


Kup ksi k Pole ksi k

```
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
  ...
  <tx:annotation-driven />

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

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

  <bean name="courseDao"
    class="com.apress.springrecipes.course.jpa.JpaCourseDao">
    <property name="jpaTemplate" ref="jpaTemplate" />
  </bean>
</beans>
```

Inną zaletą stosowania klas `HibernateTemplate` i `JpaTemplate` jest to, że przekształcają one natywne wyjątki Hibernate i JPA na wyjątki z hierarchii klasy `DataAccessException` Springa. To umożliwia spójną obsługę wyjątków we wszystkich strategiach dostępu do danych w Springu. Na przykład jeśli w trakcie utrwalania danych nastąpi naruszenie ograniczeń bazy danych, Hibernate zgłosi wyjątek `org.hibernate.exception.Constraint` ↪`ViolationException`, a w JPA będzie to wyjątek `javax.persistence.EntityExistsException`. Te wyjątki są przekształcane przez klasy `HibernateTemplate` i `JpaTemplate` na wyjątki typu `DataIntegrityViolationException` (jest to klasa pochodna od klasy `DataAccessException` Springa).

Jeśli chcesz za pomocą klasy `HibernateTemplate` lub `JpaTemplate` uzyskać dostęp do sesji platformy Hibernate albo menedżera encji JPA, aby wykonać natywne operacje platformy Hibernate lub JPA, możesz zaimplementować interfejs `HibernateCallback` lub `JpaCallback` i przekazać obiekt z tą implementacją do metody `execute()` szablonu. Dzięki temu można bezpośrednio stosować funkcje specyficzne dla danej implementacji, jeśli szablonowe mechanizmy są niewystarczające.

```
hibernateTemplate.execute(new HibernateCallback() {
  public Object doInHibernate(Session session) throws HibernateException,
    SQLException {
    // Tu można wykonać dowolne operacje — na przykład
    // unieważnić zawartość pamięci podręcznej
  }
};

jpaTemplate.execute(new JpaCallback() {
  public Object doInJpa(EntityManager em) throws PersistenceException {
    // Tu można wykonać dowolne operacje
  }
};
```

Rozszerzanie klas DAO związanych z Hibernate i JPA

Klasa DAO dla platformy Hibernate może dziedziczyć po klasie `HibernateDaoSupport`. Dzięki temu odziedziczy metody `setSessionFactory()` i `setHibernateTemplate()`. Następnie w metodach DAO wystarczy wywołać metodę `getHibernateTemplate`, aby pobrać obiekt szablonu.

```
package com.apress.springrecipes.course.hibernate;
...
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import org.springframework.transaction.annotation.Transactional;

public class HibernateCourseDao extends HibernateDaoSupport implements
```


Kup ksi k Pole ksi k

```
  CourseDao {

  @Transactional
  public void store(Course course) {
    getHibernateTemplate().saveOrUpdate(course);
  }

  @Transactional
  public void delete(Long courseId) {
    Course course = (Course) getHibernateTemplate().get(Course.class,
      courseId);
    getHibernateTemplate().delete(course);
  }

  @Transactional(readOnly = true)
  public Course findById(Long courseId) {
    return (Course) getHibernateTemplate().get(Course.class, courseId);
  }

  @Transactional(readOnly = true)
  public List<Course> findAll() {
    return getHibernateTemplate().find("from Course");
  }
}
```

Ponieważ klasa `HibernateCourseDao` dziedziczy metody `setSessionFactory()` i `setHibernateTemplate()`, możesz wstrzyknąć fabrykę sesji lub szablon Hibernate do obiektu DAO, co pozwala pobrać obiekt typu `HibernateTemplate`. Po wstrzyknięciu fabryki sesji możesz usunąć deklarację klasy `HibernateTemplate`.

```
<bean name="courseDao"
  class="com.apress.springrecipes.course.hibernate.HibernateCourseDao">
  <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

Ponadto klasa DAO związana z JPA może dziedziczyć po klasie `JpaDaoSupport`. Dzięki temu dziedziczone są metody `setEntityManagerFactory()` i `setJpaTemplate()`. W metodach DAO wystarczy wywołać metodę `getJpaTemplate()`, aby pobrać obiekt szablonu. Ten obiekt obejmuje wstępnie zainicjowany obiekt typu `EntityManagerFactory`.

```
package com.apress.springrecipes.course.jpa;
...
import org.springframework.orm.jpa.support.JpaDaoSupport;
import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao extends JpaDaoSupport implements CourseDao {

  @Transactional
  public void store(Course course) {
    getJpaTemplate().merge(course);
  }

  @Transactional
  public void delete(Long courseId) {
    Course course = getJpaTemplate().find(Course.class, courseId);
    getJpaTemplate().remove(course);
  }

  @Transactional(readOnly = true)
  public Course findById(Long courseId) {
    return getJpaTemplate().find(Course.class, courseId);
```


Kup ksi k Pole ksi k

```
  }

  @Transactional(readOnly = true)
  public List<Course> findAll() {
    return getJpaTemplate().find("from Course");
  }
}
```

Ponieważ klasa `JpaCourseDao` dziedziczy metody `setEntityManagerFactory()` i `setJpaTemplate()`, można wstrzyknąć fabrykę menedżerów encji lub szablon JPA do obiektu DAO. Jeśli wstrzykniesz fabrykę menedżerów encji, będziesz mógł usunąć deklarację obiektu typu `JpaTemplate`.

```
<bean name="courseDao"
  class="com.apress.springrecipes.course.jpa.JpaCourseDao">
  <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

15.10. Utrwalanie obiektów za pomocą sesji kontekstowych platformy Hibernate

Problem

Klasa `HibernateTemplate` Springa pozwala uprościć implementację interfejsu DAO, ponieważ zarządza sesjami i transakcjami. Jednak przy stosowaniu tej klasy obiekty DAO zależą od interfejsu API Springa.

Rozwiązanie

Zamiast klasy `HibernateTemplate` Springa można zastosować sesje kontekstowe platformy Hibernate. W Hibernate 3 takimi sesjami może zarządzać fabryka sesji. Do pobierania sesji kontekstowych służy metoda `getCurrentSession()` klasy `org.hibernate.SessionFactory`. W ramach jednej transakcji metoda `getCurrentSession()` przy każdym wywołaniu zwraca tę samą sesję. To gwarantuje, że w każdej transakcji występuje tylko jedna sesja platformy Hibernate. Opisane rozwiązanie dobrze współdziała z obsługą zarządzania transakcjami oferowaną przez Spring.

Jak to działa?

Aby zastosować sesje kontekstowe, w metodach DAO potrzebny jest dostęp do fabryki sesji. Taką fabrykę można wstrzyknąć albo za pomocą settera, albo przy użyciu argumentu konstruktora. Następnie w każdej metodzie DAO należy za pomocą fabryki sesji pobrać sesję kontekstową i wykorzystać ją do utrwalania obiektów.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.springframework.transaction.annotation.Transactional;

public class HibernateCourseDao implements CourseDao {

  private SessionFactory sessionFactory;

  public void setSessionFactory(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
  }
```


Kup ksi k Pole ksi k

```
  @Transactional
  public void store(Course course) {
    sessionFactory.getCurrentSession().saveOrUpdate(course);
  }

  @Transactional
  public void delete(Long courseId) {
    Course course = (Course) sessionFactory.getCurrentSession().get(
      Course.class, courseId);
    sessionFactory.getCurrentSession().delete(course);
  }

  @Transactional(readOnly = true)
  public Course findById(Long courseId) {
    return (Course) sessionFactory.getCurrentSession().get(
      Course.class, courseId);
  }

  @Transactional(readOnly = true)
  public List<Course> findAll() {
    Query query = sessionFactory.getCurrentSession().createQuery(
      "from Course");
    return query.list();
  }
}
```

Zauważ, że wszystkie metody DAO muszą tu być metodami transakcyjnymi. Jest to konieczne, ponieważ Spring umieszcza fabrykę typu `SessionFactory` w pośredniku, który oczekuje, że dla metod w sesji obowiązuje zarządzanie transakcjami Springa. Pośrednik próbuje znaleźć transakcję, a w razie niepowodzenia informuje, że z wątkiem nie powiązano żadnej sesji platformy Hibernate. Dlatego wszystkie metody (lub całą klasę) należy opatrzyć adnotacją `@Transactional`. To gwarantuje, że operacje utrwalające dane w metodzie DAO będą wykonywane w ramach tej samej transakcji, a więc i w tej samej sesji. Ponadto jeśli metoda komponentu z warstwy usług wywoła kilka metod DAO, dla których zastosuje własną transakcję, wszystkie te metody DAO będą wykonywane w ramach jednej sesji.

W pliku z konfiguracją ziaren dla platformy Hibernate (czyli w pliku *beans-hibernate.xml*) trzeba zadeklarować obiekt typu `HibernateTransactionManager` dla aplikacji i za pomocą elementu `<tx:annotation-driven>` włączyć deklaratywne zarządzanie transakcjami.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
  ...
  <tx:annotation-driven />

  <bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>

  <bean name="courseDao"
    class="com.apress.springrecipes.course.hibernate.HibernateCourseDao">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>
</beans>
```


Kup ksi k Pole ksi k

Pamiętaj, że klasa `HibernateTemplate` przekształca natywne wyjątki platformy Hibernate na wyjątki z hierarchii klasy `DataAccessException` Springa. To umożliwia spójną obsługę wyjątków w różnych strategiach dostępu do danych w Springu. Jednak gdy wywołujesz natywne metody w sesji platformy Hibernate, zgłaszane są wyjątki natywnego typu `HibernateException`. Jeśli chcesz, aby wyjątki platformy Hibernate były przekształcane na wyjątki z hierarchii klasy `DataAccessException` Springa, musisz dodać adnotację `@Repository` do klasy DAO, która ma zgłaszać takie wyjątki.

```
package com.apress.springrecipes.course.hibernate;
...
import org.springframework.stereotype.Repository;

@Repository
public class HibernateCourseDao implements CourseDao {
  ...
}
```

Następnie zarejestruj obiekt typu `PersistenceExceptionTranslationPostProcessor`, aby przekształcać natywne wyjątki platformy Hibernate na wyjątki z hierarchii klasy `DataAccessException` Springa. Zarejestrowany w ten sposób postprocesor przekształca wyjątki tylko dla ziaren opatrzonych adnotacją `@Repository`.

```
<beans ...>
  ...
  <bean class="org.springframework.dao.annotation.
    ↪PersistenceExceptionTranslationPostProcessor" />
</beans>
```

W Springu `@Repository` to adnotacja stereotypu. Opatrzona nią klasa komponentu jest automatycznie wykrywana za pomocą mechanizmu skanowania komponentów. W tej adnotacji można podać nazwę komponentu, a fabryka sesji zostanie automatycznie połączona w kontenerze IoC z setterem opatrzonym adnotacją `@Autowired`.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository("courseDao")
public class HibernateCourseDao implements CourseDao {

  private SessionFactory sessionFactory;

  @Autowired
  public void setSessionFactory(SessionFactory sessionFactory) {
    this.sessionFactory = sessionFactory;
  }
  ...
}
```

Następnie wystarczy zastosować element `<context:component-scan>` i usunąć pierwotną deklarację ziarna `HibernateCourseDao`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns: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/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
```


Kup ksi k Pole ksi k

```
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

  <context:component-scan
    base-package="com.apress.springrecipes.course.hibernate" />
  ...
</beans>
```

15.11. Utrwalanie obiektów za pomocą wstrzykiwania kontekstu w JPA

Problem

W środowisku Javy EE kontener może zarządzać menedżerami encji i wstrzykiwać je bezpośrednio do komponentów EJB. Komponent EJB może utrwalać dane na rzecz wstrzykniętego menedżera encji bez konieczności tworzenia takiego menedżera lub zarządzania transakcjami.

Podobnie Spring udostępnia klasę `JpaTemplate`, która upraszcza implementowanie interfejsu DAO, ponieważ zarządza menedżerami encji i transakcjami. Jednak korzystanie z tej klasy sprawia, że klasy DAO stają się zależne od interfejsu API Springa.

Rozwiązanie

Zamiast klasy `JpaTemplate` Springa można zastosować wstrzykiwanie kontekstu JPA. Główną funkcją adnotacji `@PersistenceContext` jest wstrzykiwanie menedżera encji do komponentów EJB. Spring może też interpretować tę adnotację za pomocą postprocesora ziaren. Wtedy menedżer encji jest wstrzykiwany do właściwości opatrzonej tą adnotacją. Spring gwarantuje, że wszystkie operacje utrwalania danych w ramach jednej transakcji są obsługiwane przy użyciu tego samego menedżera encji.

Jak to działa?

Aby zastosować technikę opartą na wstrzykiwaniu kontekstu, można zadeklarować pole menedżera encji w klasie DAO i opatrzyć to pole adnotacją `@PersistenceContext`. Spring wstrzyknie wtedy menedżer encji do tego pola, by możliwe było utrwalanie obiektów danej klasy.

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao implements CourseDao {

  @PersistenceContext
  private EntityManager entityManager;

  @Transactional
  public void store(Course course) {
    entityManager.merge(course);
  }
```


Kup książkę   Poleć książkę

```
  @Transactional
  public void delete(Long courseId) {
    Course course = entityManager.find(Course.class, courseId);
    entityManager.remove(course);
  }

  @Transactional(readOnly = true)
  public Course findById(Long courseId) {
    return entityManager.find(Course.class, courseId);
  }

  @Transactional(readOnly = true)
  public List<Course> findAll() {
    Query query = entityManager.createQuery("from Course");
    return query.getResultList();
  }
}
```

Do każdej metody DAO (lub całej klasy DAO) można dodać adnotację `@Transactional`, aby te metody działały w ramach transakcji. To gwarantuje, że operacje związane z utrwalaniem danych w metodzie DAO będą wykonywane w ramach tej samej transakcji i przy użyciu tego samego menedżera encji.

W pliku z konfiguracją ziaren JPA (czyli w pliku *beans-jpa.xml*) trzeba zadeklarować obiekt typu `JpaTransactionManager` i za pomocą elementu `<tx:annotation-driven>` włączyć deklaratywne zarządzanie transakcjami. Na potrzeby wstrzykiwania menedżera encji do właściwości opatrzonych adnotacją `@PersistenceContext` należy zarejestrować obiekt typu `PersistenceAnnotationBeanPostProcessor`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  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/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">
  ...
  <tx:annotation-driven />

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

  <bean name="courseDao"
    class="com.apress.springrecipes.course.jpa.JpaCourseDao" />

  <bean class=
    "org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor" />
</beans>
```

Jeśli ustawiony jest element `<context:annotation-config>`, obiekt typu `PersistenceAnnotationBean↪PostProcessor` jest rejestrowany automatycznie. Można więc usunąć jawną deklarację ziarna.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns: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/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
```


Kup ksi k Pole ksi k

```
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd">

  <context:annotation-config />
  ...
</beans>
```

Ten postprocesor ziaren potrafi też wstrzyknąć fabrykę menedżerów encji do właściwości opatrzonej adnotacją `@PersistenceUnit`. Dzięki temu można samodzielnie tworzyć menedżery encji i zarządzać transakcjami. To rozwiązanie nie różni się od wstrzykiwania fabryki menedżerów encji za pomocą settera.

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;

public class JpaCourseDao implements CourseDao {
  @PersistenceContext
  private EntityManager entityManager;

  @PersistenceUnit
  private EntityManagerFactory entityManagerFactory;
  ...
}
```

Pamiętaj, że klasa `JpaTemplate` przekształca natywne wyjątki JPA na wyjątki z hierarchii klasy `DataAccessException` Springa. Jednak przy wywoływaniu natywnych metod za pomocą menedżera encji JPA zgłaszane będą wyjątki natywnego typu `PersistenceException` lub inne wyjątki Javy SE (na przykład `IllegalArgumentException` i `IllegalStateException`). Jeśli chcesz, aby wyjątki JPA były przekształcane na wyjątki z hierarchii klasy `DataAccessException` Springa, musisz dodać adnotację `@Repository` do klasy DAO.

```
package com.apress.springrecipes.course.jpa;
...
import org.springframework.stereotype.Repository;

@Repository("courseDao")
public class JpaCourseDao implements CourseDao {
  ...
}
```

Następnie zarejestruj obiekt typu `PersistenceExceptionTranslationPostProcessor`, aby przekształcał natywne wyjątki JPA na wyjątki z hierarchii klasy `DataAccessException` Springa. Możesz też ustawić element `<context:component-scan>` i usunąć deklarację ziarna `JpaCourseDao`, ponieważ w Springu 2.5 i w nowszych wersjach `@Repository` to adnotacja stereotypu.

```
<beans ...>
  ...
  <context:component-scan
    base-package="com.apress.springrecipes.course.jpa" />

  <bean class=
    "org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />
</beans>
```


Kup ksi k Pole ksi k

Podsumowanie

W tym rozdziale opisaliśmy, jak korzystać z dostępnej w Springu obsługi technologii JDBC, Hibernate i JPA. Dowiedziałeś się, jak skonfigurować obiekt typu `DataSource` w celu połączenia się z bazą danych i jak używać klas `JdbcTemplate`, `HibernateTemplate` i `JpaTemplate` Springa, aby nie trzeba było pisać powtarzającego się kodu. Zobaczyłeś, jak korzystać z narzędziowych klas bazowych do tworzenia klas DAO na potrzeby technologii JDBC, Hibernate i JPA. Wiesz już też, jak używać w Springu adnotacji stereotypów i skanowania komponentów w celu łatwego tworzenia nowych klas DAO i usług za pomocą niewielkiej ilości kodu w XML-u.

Z następnego rozdziału nauczysz się stosować w Springu transakcje (dla interfejsu JMS i baz danych), aby zapewnić spójny stan usług.

Kup ksi k

Pole ksi k

Kup ksi k

Pole ksi k

Skorowidz

A

ACL, Access Control List, 192
adapter, 714
  interfejsu użytkownika, 730
  kanałów, 730
  wiadomości przychodzących, 731
adaptery trzeciego rodzaju, 730
adnotacja
  @Aspect, 122
  @Autowired, 57, 89, 276, 306
  @Before, 122
  @Configurable, 150
  @Configuration, 89
  @ContextConfiguration, 476
  @Controller, 268, 273, 301
  @DeclareParents, 137
  @DependsOn, 91
  @DirtiesContext, 476
  @Endpoint, 653
  @Entity, 552
  @ExpectedException, 491
  @Gridify, 787–791
  @Header, 717
  @IfProfileValue, 491
  @Interceptors, 622
  @Lazy, 91
  @ManagedAttribute, 665
  @ManagedOperation, 665
  @ManagedResource, 665
  @NotNull, 319
  @Order, 129
  @PathVariable, 330
  @Pattern, 319
  @PayloadRoot, 654
  @PersistenceContext, 570
  @PersistenceUnit, 572
  @Pointcut, 130
  @PostConstruct, 87, 811
  @PreDestroy, 87
  @Primary, 91
  @Qualifier, 60
  @Remote, 622
  @Repeat, 491
  @Repository, 569
  @RequestingMapping, 507
  @RequestMapping, 268, 273, 330
  @Required, 51
  @Resource, 57
  @Scheduled, 683
  @ServiceActivator, 715
  @Timed, 491
  @Transactional, 485, 564, 590, 700
  @Transformer, 394
  @Value, 296
  @WebMethod, 631
  @XmlRootElement, 333
adnotacje
  dotyczące testów, 491
  języka AspectJ, 120, 122
  JPA, 551
  OSGi, 831
adres URL, 163, 170
agent wplatania, 146
agregatory, 726, 728
Ajax, 262
akcja, 242, 260
akcja formularza, 239
akcje Strutsa, 217
aktualizowanie
  bazy danych, 526
    instrukcje SQL-a, 528
    klasa SimpleJdbcTemplate, 537
    kreator instrukcji, 526
    masowe, 529
    setter instrukcji, 528
  ziaren, 160

Kup książkę   Poleć książkę

aktywowanie skanowania adnotacji, 273
algorytmy szyfrujące, 176
AMF, Action Message Format, 358
anulowanie transakcji, 580, 762
Apache ActiveMQ, 390, 689
Apache Ant, 405
Apache CXF, 631
Apache Derby, 250
Apache Ivy, 405
Apache James Server, 673
Apache Maven, 439
Apache OpenEJB, 614
Apache Pluto, 501, 503
Apache POI, 321
Apache Portals, 501
Apache Struts, 203
Apache Tiles, 454
Apache Tomcat, 409, 627
Apache XMLBeans, 638
aplikacje
  Fleksa, 360
  JSF, 218
  Grails, 404
  MVC, 165
  sieciowe, 172, 206, 213, 247
  Spring MVC, 231, 271
  Spring-WS, 641
  Strutsa, 214
appender, 423
aspekt
  AnnotationBeanConfigurerAspect, 150
  AnnotationTransactionAspect, 605
asynchroniczne odbieranie komunikatów, 701
atrapa, mock, 463, 466
atrybut, attribute, 665
  auto-config, 164
  destroy-action, 827
  destroy-method, 87
  filter-name, 212
  init-method, 87
  offerdate, 435
  order, 293
  propagation, 594, 595
  propagation transakcji, 591
  refresh-check-delay, 160
  script-source, 161
  url, 378
atrybuty usług, 828
automatyczne
  łączenie ziaren, 53–62
  skanowanie komponentów, 67
  wykrywanie ziaren, 666
autoryzacja, 163

B

baza danych, 177, 519, 577
  PostgreSQL, 752
  HSQLDB, 416
bezpieczeństwo, 163
biblioteka
  Activation, 674
  Active MQ, 390
  Ajax4JSF, 262
  Apache POI, 321
  CGLIB, 120
  EasyMock, 470
  Facelets, 256
  Hibernate 3, 250
  iText, 321
  Javassist, 250
  Jaxen, 355
  JAXP 1.3, 355
  JBoss EL, 231
  JDOM, 338
  JExcelAPI, 321
  JLine, 439
  JSTL, 267
  Project Rome, 338, 339, 356
  RichFaces, 262, 265
  SWF Object JavaScript, 364
  znaczników formularzy, 513
  znaczników JSP, 190
biblioteki współużytkowane, 833
błędy, 723
BMT, Bean-Managed Transaction, 575
BPM, Business Process Management, 798
brama, 740
broker
  Apache ActiveMQ, 689
  encji GraniteDS, 386
  komunikatów, 388, 692

C

CICS, Customer Information Control System, 747
CMT, Container-Managed Transaction, 575
CRM, Customer Relationship Management, 448
CRUD, create, read, update, delete, 414
cykl
  życia EJB, 616
  życia encji, 551
czas
  kompilacji, 143
  ładowania, 143

Kup ksi k   Pole ksi k

D

dane
  o użytkownikach, 184
  uwierzytelniające, 163
  wejściowe, 753, 758
  wyjściowe, 755
DAO, Data Access Object, 66, 521
data, 325
DBCP, Database Connection Pooling Services, 524
definiowanie
  kolekcji, 47, 48
  kontraktu usługi sieciowej, 636
  stanów
    akcji, 242
    końcowych, 244
    podejmowania decyzji, 243
    podprzepływów, 246
    widoku, 239
  szablonu e-maili, 676
  zdarzeń, 103
deklaratywne zarządzanie transakcjami, 564, 568, 575, 588, 590
deklarowanie
  akcji Strutsa, 217
  aspektów, 122, 140, 141
  punktów przecięcia, 135, 141
  rad, 142
  serwletu, 207
  wprowadzeń, 142
  ziaren, 27, 74
    wewnętrznych, 42
    zarządzanych, 223
deserializacja, unmarshalling, 610
deskryptor wdrażania, 205
diagram
  obiektów, 65
  przepływu sterowania, 235, 242, 244, 247, 254
dodawanie
  operacji do ziaren, 136
  stanu do ziarna, 138
  wiadomości, 736
dokument PDF, 324
DOM, Document Object Model, 349
domeny, 689
domyślna
  konfiguracja, 364
  lokalizacja, 696
dostawca AclEntryAfterInvocationProvider, 199
dostawcy uwierzytelniania, 176
dostęp do
  adresów URL, 163, 170
  bazy danych, 483, 487, 519, 579
  danych JSON, 348
  komponentów EJB 2.x, 618
  komponentów EJB 3.0, 623
  kontekstu aplikacji, 215, 474–477
  kontenera, 793
  kontrolera, 297
  parametrów, 814
  repozytoriów, 822
  Springa, 204
  usług typu REST, 333, 348, 356
  zdalnych ziaren, 670, 671
DRY, Don't Repeat Yourself, 400
DSL, Domain-Specific Language, 19
DTP, Data Tools Platform, 520
DWR, Direct Web Remoting, 203
dynamiczne
  jednostki pośredniczące, 137
  uzupełnianie strony, 364
działanie trybu
  REQUIRED, 594
  REQUIRES_NEW, 595
dziedziczenie, 429
dziedziczenie konfiguracji ziarna, 62

E

EAI, Enterprise Application Integration, 709
EDA, Event-Driven Architecture, 388
edytor właściwości, 105, 108
egzemplarz
  kontekstu aplikacji, 25
  kontenera IoC, 23
eksportowanie
  aplikacji Grails, 408
  usług, 825
  ziaren, 383, 657, 662
ekstendery, 831
element, *Patrz także* znacznik
  dataSource, 421, 422
  decision, 768
  errors, 312
  gateway, 743
  osgi:reference, 827
  osgi:service, 825
  poller, 733
  router, 729
  service, 830
  tasklet, 761
  tx:attributes, 589
elementy, items, 344
e-mail, 672
encja Customer, 779
ESB, Enterprise Service Bus, 391, 709
Excel, 321, 323

F

fabryczna metoda egzemplarza, 72
fabryka
    menedżerów encji, 554, 556, 560
    sesji, 549
    sesji Hibernate, 556
    ziaren, 23
fasada, 740
filtr, 208, 210
filtr DelegatingFilterProxy, 248
filtrowanie
    skanowanych komponentów, 69
    żądań HTTP, 170
Flash, 359
Flex, 357, 398
Flex Builder, 362, 364
format
    AMF, 372, 384
    Atom, 338, 339
    JAXB, 631
    JSON, 345, 346
    MXML, 360
    PDF, 321, 325
    RSS, 338, 339
    SOAP, 370
    XML, 348
    XLS, 321, 325
formater, 149
formularz, 173, 215, 240, 298, 510
formularz kreatora, 310, 316
funkcja
    anonimowa, 361
    playTrack, 367
    resultHandler, 384

G

generowanie
    kontrolerów CRUD, 414
    nazw, 69
    plików
        PDF, 321
        WSDL, 639
        XLS, 321
        XSD, 637
GORM, Groovy Object Relational Mapper, 420, 432
Grails, 403, 410

H

hasło pluto, 502
hierarchia ziaren, 65
HQL, Hibernate Query Language, 432, 519

I

identyfikator SID, 192
implementacja
    filtra, 211
    interfejsu DAO, 465, 522
    interfejsu ItemReader, 756
    interfejsu ItemWriter, 757
    jednostki ładującej, 794
    menedżera transakcji, 581
    usługi sieciowej, 640
    ziarna, 153
    źródła danych, 524
importowanie projektu, 446, 447
informacje
    o błędach, 299
    o punkcie złączenia, 127
    o uwierzytelnieniu, 190
inicjowanie
    obiektu, 302
    ziarna, 84, 87
instalowanie
    aplikacji sieciowej, 278
    GridGain, 786
    JPA, 449
    klienta, 824
    obsługi integracji, 374
    platformy Grails, 403
    plików .war, 832
    portletu, 508
    systemu jBPM, 802
    środowiska Equinox, 819
    Terracotty, 782
    wtyczki Clojure, 409
instrukcje SQL-a, 528
integrowanie Springa
    z platformą DWR, 223
    z platformą Grails, 409
    z platformą JSF, 218, 255
    z platformą Struts, 212
    z systemem jBPM, 804
integrowanie systemów, 710, 712, 718
interceptor, 326
interceptory przetwarzania, 282, 285
interfejs
    API CommonJ, 112
    API Hibernate, 551
    API JavaMail, 673
    API Timer, 112
    ApplicationContext, 24, 92
    ApplicationContextAware, 477
    ApplicationEventPublisherAware, 104
    ApplicationListener, 103
    BeanFactory, 24
    BeanPostProcessor, 96

Kup ksi k   Pole ksi k

CustomerService, 811
DAO, 66, 465, 522
DataSource, 523
DisposableBean, 84, 86
EntityManager, 548
Executor, 109
ExecutoreService, 110
ExpressionParser, 80
FactoryBean, 45
HandlerExceptionResolver, 294
HttpConverterMessage, 349
InitializingBean, 84, 86, 98
InterestCalculator, 154
ItemReader, 756
ItemWriter, 756, 757
JNDI, 45
JobRepository, 749
JPA, 250
JTA, 581
MailSender, 675
menedżera transakcji, 583
MessageHeaders, 716
MessageSource, 731, 736
MessageSourceAware, 102
MimeMessagePreparator, 678
PlatformTransactionManager, 582
PreparedStatementCreator, 527
PriorityOrdered, 98
RowCallbackHandler, 531
RowMapper, 531
Serializable, 237, 510
Servlet API, 206
Session, 548
SessionBean, 616
SqlParameterSource, 541
TaskExecutor, 109, 112, 117
Validator, 307
WorkManager, 112
ITD, Inter-Type Declaration, 438
izolowanie transakcji, 596, 597

J

Java 1.5, 537
Java EE, 112, 271, 657
Java SE, 109
JAXB, Java Architecture for XML Binding, 631
JAX-WS, 630
jBPM, 797
JCA, Java Cryptography Architecture, 112
JCA, Java EE Connector Architecture, 741
JDK, Java Development Kit, 110
JDK Timer, 657
język
ActionScript 3.0, 359
AspectJ, 119
BeanShell, 153, 156, 159
BPEL, 800
Groovy, 153, 403
HQL, 432, 519
JRuby, 153, 158
OGNL, 77, 231
SpEL, 77, 296, 774
Unified EL, 77, 231
WADL, 334
WSDL, 628
XPath, 351
języki
skryptowe, 153
wyrażeń, 77, 78
JMS, Java Message Service, 687
JMX, Java Management Extensions, 657
JSF, JavaServer Faces, 203, 222
JSON, JavaScript Object Notation, 345
JTA, Java Transaction API, 581

K

kanał
Atom, 338
errorChannel, 723
input-channel, 393
output-channel, 395
requests, 742
RSS, 338
kanały
błędów, 725
informacyjne, 338
kardynalność, 829
klasa
AbstractFactoryBean, 46
AbstractStatelessSessionBean, 616
AbstractTransactionalJUnit4SpringContextTests, 486
AclEntryVoter, 197
ActionScriptu, 361
Activator, 817, 820
ActiveMQTopic, 691
AnnotationMBeanExporter, 666
ApplicationEvent, 103
BackOfficeImpl, 691
BusinessConfiguration, 91
CastorMarshaller, 653
CityServiceRequestAuditor, 210
CometEngine, 388
ComplexFormatter, 150
Configuration, 552
CookieLocaleResolver, 286
CronTriggerBean, 682
CustomDateEditor, 106
DataAccessException, 543, 565, 572
DelegatingFilterProxy, 171

Kup ksi k
Pole ksi k

EmailErrorNotifier, 675
Environment, 412
ExternalInterface, 366
FlatFileItemReader, 754
FrontDeskImpl, 689
FrontDeskMain, 692
HibernateTemplate, 562, 569
HTTPService, 368
Item, 385
JavaMailSenderImpl, 675
JdbcDaoSupport, 536
JdbcTemplate, 537
JmsGatewaySupport, 697
JobDetailBean, 682
JpaTemplate, 562
MBeanExporter, 658
Member, 332
MessageListenerContainer, 707
MethodInvokingJobDetailFactoryBean, 682
MimeMessage, 678
Product, 34
ProductCreator, 72
RedirectView, 501
RestTemplate, 334, 349
RetryTemplate, 763
RssFeedView, 343
SimpleJdbcTemplate, 537, 538
SimpleJobRepository, 751
SimpleMessagingGateway, 741
SimpleTriggerBean, 682
StreamSource, 349
TestCase, 459
WelcomeController, 407
XPath, 352
XPathTemplate, 356
klastry, 777
klasy
DAO, 697
domenowe, 412, 414, 416
encji, 547
niezależne, 463
platformy TestContext, 473, 482
punktów końcowych, 641
serializujące, 649
wyjątków, 543
zależne, 466
klucz biznesowy, 812
kodowanie, 350
kod błędu, 544
kolejka ActiveMQ, 687, 688
kolekcja Properties, 32
kolekcje niezależne, 48
kompilator
ajc, 143
compc, 363
mxmlc, 363
komponent
reader, 758
router, 729
komponenty
EJB 2.x, 613, 616, 618
EJB 3.0, 621, 623
komunikat, 637, 687
JMS, 382, 688, 698
o błędach, 724, 725
tekstowy, 100
konfigurowanie
aspektów AspectJ, 148
bazy danych, 520
fabryk zasobów ORM, 556
fabryki menedżerów encji, 560
fabryki sesji, 557
infrastruktury Spring Batch, 749
JPA, 251, 252
kolekcji, 29
kroku, 763
narzędzia GridGain, 794
niestandardowych appenderów, 423
platformy DWR, 226
platformy RichFaces, 263
przepływu, 249
relacyjnych baz danych, 421
Roo, 439
serwletu JSF, 256
Terracotty, 783
transformera, 721
układów, 424
usług, 193
w XML-u, 140
ziaren, 26, 36, 62, 99, 205, 217
źródła danych, 523
kontekst aplikacji, 23, 805
kontener
ActionScript, 398, 402
IoC, 23, 71
portletów, 502
kontrakt, 334
danych, 636
usługi, 636
usługi sieciowej, 636
kontrola dostępu, 163, 185, 197
kontroler, 167, 267
formularza, 300, 510, 512
formularza kreatora, 310, 313
portletów, 501
Reservation FormController, 315
kontrolowanie wykonywania kroków, 765
korporacyjna magistrala usług, 709
koszty generowania obiektu, 535
koszyk zakupów, 83
kreator instrukcji, 526
krok, 765
kroki sekwencyjne, 766

Kup ksi k Pole ksi k

L

liczby zespolone, 144
lista, 29
lista evaluateAsNodeList, 353
listy ACL, 192, 196, 197
localhost, 378
logowanie, 172, 173
logowanie anonimowe, 164, 175
lokalizacja szablonu, 289

Ł

ładowanie, 143
łączenie
  klas, 336
  procesorów, 760
  wyrażeń, 134
  z bazą, 521
  ziaren, 53
    automatyczne, 53–62
    na podstawie konstruktora, 55
    na podstawie nazwy, 55, 62
    na podstawie typu, 54, 60
    z adnotacją @Autowired, 57
    z adnotacją @Resource, 57
    z automatycznym wyborem trybu, 56
    zgodnych typów, 57, 59

M

magazyny danych, 753
magistrala ESB, 391, 711
manipulowanie encjami, 452
mapowanie obiektowo-relacyjne, 417
Maven, 439
MDB, Message-Driven Bean, 701
MDP, Message-Driven POJO, 712
mechanizm
  GORM, 420
  HTTP Invoker, 628
  szeregowania, 332
  wplatania, 147
menedżer
  encji, 556
  podejmowania decyzji, 185
  transactionManager, 591, 761
  transakcji, 582, 583
metadane, 339, 665, 748
metoda
  addCallback(), 367
  afterCompletion(), 282
  afterPropertiesSet(), 84
  aspectOf(), 148
  bid(), 383
  buildFeedMetadata(), 342
  call(), 366
  commit(), 580
  convertAndSend(), 698
  createEntityManagerFactory(), 560
  createProduct(), 71
  destroy(), 84
  ejbCreate(), 616
  equals(), 31
  evaluate(), 355
  getAsText(), 108
  getAtomFeed(), 340
  getBean(), 25, 82
  getForObject(), 349, 355
  getItemsForAuction(), 383
  getMessage(), 102
  getOrder(), 129
  getPath(), 96
  getResource(), 93, 94
  getRSSFeed(), 340
  getSequence(), 26
  marshalSendAndReceive(), 652
  onApplicationEvent(), 103
  postProcessAfterInitialization(), 96
  postProcessBeforeInitialization(), 96
  receiveAndConvert(), 698
  replicate(), 659
  rollback(), 580
  saluteSomeoneInForeignLanguage(), 787
  sendMail(), 691
  setDataSource(), 536
  setJdbcTemplate(), 536
  submit(), 110
  submitForm(), 301
  swfobject.embedSWF(), 365
  toString(), 144, 150
  tweet(), 738
  validate(), 316
  verify(), 470
metody
  fabryczne, 71
  klasy RestTemplate, 335
  transakcyjne, 591
model, 267
  BPM, 801
  Comet, 387
  DOM, 349
modele przepływu pracy, 800
modelowanie przepływów, 236
moduł
  ACL, 192
  DBCP, 524
modyfikator
  public, 131
  synchronized, 26, 165

Kup ksi k   Pole ksi k

modyfikowanie
adresów URL, 326
nagłówków komunikatów, 722
obsługi wyjątków, 545
treści komunikatu, 721
MOM, Message-Oriented Middleware, 387
MVC, Model-View-Controller, 267

N

nadawca, 102
nadmiarowość, 775
nagłówek Accept, 291, 293
nagłówki komunikatów, 717, 722
namiastka, stub, 463, 466
narzędzia firmy SpringSource, 834
narzędzie
ActiveMQ, 692
AJDT, 454
Apache Ant, 405
Apache CXF, 631
Apache Ivy, 405
Apache Maven, 439
Apache Tiles, 454
Apache Tomcat, 409, 627
Apache XMLBeans, 638
Axway Integrator, 711
Eclipse Equinox, 816
Ehcache, 184
GridGain, 785, 792, 794
Hibernate, 409
ij, 576
Java ORM Hibernate, 432
JConsole, 662
Log4J, 422
OpenEJB Standalone Server, 614
ORM, 525
Quartz Scheduler, 679
RabbitMQ, 687
Spring Batch, 739
Spring BlazeDS Integration, 374, 387, 396
Spring Roo, 437, 440
Terracotta, 777
nawiasy klamrowe, 331
nawiązywanie połączeń, 706
nazwa
widoku, 289
ziarna akcji, 217
negocjowanie treści, 291, 326
niestandardowe
appendery, 423
edytory właściwości, 108
implementacje interfejsów, 756
szablony, 429
układy, 429
walidatory, 414
znaczniki, 434
niezgodność impedancji, 630

O

obiekt
docelowy, 139
this, 139
typu Complex, 146
typu HTTPService, 369
typu JobExecution, 749
typu JobInstance, 748, 749
typu JobLauncher, 769
typu JobParameters, 748, 773
typu JobRepository, 750
typu Object, 384
typu Runnable, 113
typu StepExecution, 749
obiekty
domenowe, 149, 196, 199
JNDI, 624
MDP, 713
po stronie serwera, 384
POJO, 703
POJO sterowane komunikatami, 702, 712
obsługa
adnotacji AspectJ, 120
błędów, 723
formularzy, 297
formularzy z portletów, 510
integracji, 374
JUnit 3, 474, 484, 488
kontrolerów, 273
narzędzia Roo, 446
obiektów domenowych, 199
odwzorowań, 519
pamięci podręcznej, 184
poczty elektronicznej, 672
punktów końcowych JAX-WS, 631
Quartza, 679
systemu plików, 719
testów, 491
transakcji, 197, 761
usług JMS, 387
wielu formularzy, 310
współbieżności, 109
wyjątków, 542, 545, 565
zagadnień przecinających, 122
oczekiwanie na zdarzenia, 104
odbieranie komunikatów JMS
domyślna lokalizacja, 696
obiekt POJO, 703
odbiornik, 702
operacje, 688
szablon, 693

Kup ksi k Pole ksi k

odbieranie powiadomień JMX, 669
odbiorca, 102
odbiornik
ActionListener, 260
HibernateFlowExecutionListener, 250
JpaFlowExecutionListener, 250
komunikatów JMS, 700, 702
odpytywanie, polling, 387
odwzorowania, 31
obiektowo-relacyjne, 519
ORM, 547
ograniczenie CHECK, 577, 580
określanie
typu danych, 43
ustawień regionalnych, 285
widoków na podstawie
kilku klas, 290
lokalizacji szablonu, 289
nazw, 289
pakietu zasobów, 290
pliku konfiguracyjnego, 289
operacje, operations, 136, 665
operacje CRUD, 414–416
operatory porównywania, 433
optymalizowanie pliku XSD, 638
ORM, Object/Relational Mapping, 417, 519
OSGi, Open Service Gateway initiative, 815
OXM, Object/XML Mapping, 649

P

pakiet
Adobe Creative Suite, 363
Dojo JavaScript, 256
JDK, 109
SDK, 363
STS, 234, 834
pakiety zasobów, 101, 290
pakunek, bundle, 815
pakunek klienta, 819
parametr
classpath, 65, 93
FlashVars, 364
parametry
nazwane, 540
punktów przecięcia, 135
w zadaniach, 772
parser języka SpEL, 80
pętla for each, 31
piaskownica, sandbox, 364
pierwszeństwo
aspektów, 128
rad, advice precedence, 119
piggybacking, 387
planowanie zadań, 657
narzędzie Quartz Scheduler, 679
przestrzeń nazw Scheduling, 683
platforma
Apache Struts, 203
AspectWerkz AspectJ, 122
CXF, 633, 635
DWR, 223, 226
Flex, 359
klasa ExternalInterface, 366
obsługa protokołu HTTPS, 368
parametry FlashVars, 364
piaskownica, 364
środowisko Flex Builder, 362
tworzenie aplikacji, 360
usługi SOAP, 370
wysyłanie komunikatów, 396
wywołania zdalne, 372
Grails, 403, 410
appender, 423
eksportowanie aplikacji, 408
klasy domenowe, 412
operacje CRUD, 415
rejestrator, 423
rejestrowanie danych wyjściowych, 422
relacyjna baza danych, 421
struktura plików, 405
system pamięci trwałej, 420
szablony, 429
środowisko uruchomieniowe, 410
testowanie, 424, 428
tworzenie aplikacji, 406
typy układów, 424
układy, 429
umiędzynarodawianie, 418
uruchamianie aplikacji, 406
wtyczki, 408
zapytania GORM, 432
znaczniki niestandardowe, 434
Hibernate, 382, 548
Java Activation Framework, 293
JSF, 218
JUnit 3, 457, 459
Quartz, 116
RichFaces, 262, 263
Spring Batch, 747
infrastruktura, 749
komponent reader, 758
mechanizm ponawiania prób, 762
obsługa transakcji, 761
skalowalność, 752
uruchamianie zadania, 769
współbieżność, 766
zdalna obsługa porcji, 767

Kup ksi k

Pole ksi k

platforma
Spring Dynamic Modules, 821
dostosowywanie działania, 831
eksportowanie usług, 825
Spring Faces, 256, 261
Spring Integration, 709
adapter kanałów, 730
informacje o kontekście, 716
integrowanie systemów, 718
obiekty MDP, 713
obsługa błędów, 723
przekształcanie komunikatów, 720
ukrycie obsługi komunikatów, 742
Spring JavaScript, 261
Spring JDBC, 542
obsługa wyjątków, 542
Spring MVC, 165, 267, 271
adnotacja @RequestMapping, 278
adnotacja @Value, 297
formularze wielostronicowe, 310
generowanie plików PDF, 321
generowanie plików XLS, 321
interceptory przetwarzania, 282
konfigurowanie aplikacji, 271
kontrolery, 274
negocjowanie treści, 291
obsługa formularzy, 297
skanowanie adnotacji, 273
standard JSR-303, 319
testowanie kontrolerów, 471
ustawienia regionalne, 285
walidatory, 307
wiązanie wyjątków, 294
wiązanie żądań, 279
widoki JSP, 277
wygasanie danych, 309
Spring Portlet MVC
formularze, 510
kontrolery, 499
przepływ sterowania, 496
tworzenie portletu, 495
wiązanie żądań na podstawie parametru, 508
wiązanie żądań z metodami, 503
wiązanie żądań z trybem portletu, 505
Spring Roo, 437
konfigurowanie środowiska, 439
środowisko STS, 445
testowanie, 450
tworzenie projektu, 441
usuwanie z projektu, 454
wydajność, 448
Spring Security, 163, 171
Spring Web Flow, 229, 252
integrowanie z technologią JSF, 255
modelowanie przepływów, 236
platforma RichFaces, 262
przepływ sterowania, 229
utrwalanie obiektów, 249
zabezpieczanie przepływów sterowania, 247
Spring-WS, 640, 645
Struts 1.x, 212
TestContext, 473–492
adnotacje, 491
TestNG, 457
platformy testowe, 457
plik
_reservationList.gsp, 431
about.jsp, 296
aop.xml, 146
application.properties, 405
applicationContext.xml, 223, 511
bashrc, 404
batch.xml, 751
beanRefContext.xml, 622
beans-back.xml, 692
beans-ejb.xml, 617
beans-front.xml, 692
beans-hibernate.xml, 557, 563, 568
beans-jpa.xml, 571
bookingForm.jsp, 513
bookingSuccess.jsp, 514
borrowForm.xhtml, 258
borrowReview.xhtml, 259
build.xml, 405
bundlecontext.xml, 823
bundle-osgi-context.xml, 825
client.xml, 647
Config.groovy, 411, 422
config.ini, 824
Course.hbm.xml, 548
court.iml, 405
court.ipr, 405
court.iws, 405
court.launch, 405
court.tmproj, 405
court-service.xml, 272
court-servlet.xml, 288
court-test.launch, 405
crossdomain.xml, 368
Customer_Roo_ Configurable.aj, 450
customerconsole-context.xml, 784
DataSource.groovy, 411, 420
distance.jsp, 221
dwr.xml, 225, 226
ejb-jar.xml, 616
error.jsp, 295
faces-config.xml, 257
flightHelp.jsp, 507
flight-portlet.xml, 507
hibernate.cfg.xml, 549, 557

introduction.jsp, 235
ivy.xml, 406
ivysettings.xml, 406
jbpm.cfg.xml, 808
jbpm4.properties, 806
library-service.xml, 251
library-servlet.xml, 233
library-webflow.xml, 239, 249, 257
main.gsp, 430
MANIFEST.MF, 818, 827
mapping.xml, 651
message.properties, 296
messages.properties, 308, 516
persistence.xml, 556
Player.groovy, 413
pom.xml, 443, 455
portlet.xml, 497, 504
RegisterCustomer.jpdl.xml, 812
Reservation.groovy, 413
reservationQuery.jsp, 277
response.xml, 637
schema-postgresql.sql, 750
sql-error-codes.xml, 544
stacktrace.log, 423
struts-config.xml, 216
template.xhtml, 258
views.properties, 325
views.xml, 289
weather-servlet.xml, 626
web.xml, 210, 220
welcome.jsp, 284
pliki
.as, 359
.gar, 792, 793
.jar, 165, 178, 443, 818, 822
.mxml, 360, 400
.pdf, 326
.properties, 418
.roo, 442
.swf, 359
.war, 374, 408, 793, 832
.wsdl, 632
.xls, 326
.xml, 333
.xsd, 629
konfiguracyjne, 166
konfiguracyjne XML, 27
X.as, 387
pobieranie
danych, 349, 353
danych z bazy, 530
interfejs RowCallbackHandler, 531
interfejs RowMapper, 531
klasa SimpleJdbcTemplate, 538
pojedyncze wartości, 534
wiersze, 532
komunikatów, 388, 689
pośrednika bramy, 744
ziaren, 25
podawanie referencji, 39
podejmowanie decyzji, 243
podział zdarzeń, 739
POJO, 23
pola statyczne, 74
polecenie
grails run-app, 408, 415
grails test-app, 425
grails war, 408
hint, 448
hint controllers, 452
połączenie z bazą danych, 178
ponawianie prób, 762–765
POP3, 675
port
1099, 664
4555, 673
portlety, 495
postprocesor, 85
postprocesor ziaren, 51, 96, 99, 572
pośrednik
bramy, 744
EJB, 618, 621
MBeanProxy, 671
powiadomienia JMX, 667
powłoka Roo, 441, 442, 443
poziom izolacji transakcji, 597
READ_COMMITTED, 598
READ_UNCOMMITTED, 598
REPEATABLE_READ, 600
SERIALIZABLE, 602
w interfejsach API, 602
w pośrednikach, 602
w radach transakcji, 602
poznawanie zasobów, 92
prawo Amdahla, 776
proces, 798
proces biznesowy, 810, 812
programowanie aspektowe, 119, 765
programowe zarządzanie transakcjami, 583, 585
projekt
Apache Active MQ, 390, 689
FlexMojos, 363
Java Config, 89
Mavena, 443
Spring ActionScript, 398
Spring Batch, 739
Spring Integration, 391
spring-plugins, 284
propagacja
w trybie REQUIRED, 593
w trybie REQUIRES_NEW, 595

Kup ksi k Pole ksi k

propagacja transakcji, 592
protokół
    AMF, 358, 373
    HTTP, 625
    HTTPS, 368
    JMXMP, 664
    LDAP, 182
    POP3, 675
    SMTP, 673
    XMPP, 745
przebieg obsługi żądań, 268
przechowywanie
    konfiguracji ziarna, 99
    stanu obiektów, 777
przechwytywanie żądań, 282
przedrostek redirect, 291
przedrostki ścieżek, 95
przekształcanie
    komunikatów, 716, 720
    komunikatów JMS, 698, 704
    łańcuchów znaków, 106
    typów danych, 108
przepływ
    pracy, 799, 800
    sterowania, 229, 237, 249, 760
przestrzeń
    nazw osgi, 823
    nazw Scheduling, 683
przetwarzanie
    adnotacji OSGi, 831
    danych wejściowych, 758
    formularzy, 300
    listy, 353
    rozproszone, 775
    równoległe, 776, 790
    w gridzie, 785
    wsadowe, 747
    wyrażeń, 77
    ziaren, 222
    żądania, 282
przypadki testowe, 459
przypisywanie wartości, 296
publikowanie
    danych, 338, 345
    komunikatów, 707
    pliku WSDL, 644
    powiadomień JMX, 668
    usług, 830
    usług typu REST, 329
    zdarzeń, 104
punkt
    końcowy, 640
    końcowy z obsługą serializacji, 650
    końcowy usługi, 642
    przecięcia, 130, 141, 189
    złączenia, join point, 123, 127

Q

Quartz Scheduler, 657, 679

R

rada, 122
rada transakcji, 588, 595
rady
    typu After, 124
    typu AfterReturning, 124
    typu AfterThrowing, 125
    typu Around, 126
    typu Before, 122
ranking, 828
referencje do ziaren, 39–41, 90
reguły wycofywania transakcji, 603
rejestr OSGi, 828
rejestrator, logger, 423
rejestrowanie
    danych wyjściowych, 422
    edytorów właściwości, 105
    portletów, 502
    postprocesora, 98
    ziaren MBeans, 660
relacyjna baza danych, 420, 428
repozytorium
    LDAP, 181
    SpringSource Enterprise Bundle Repository, 822
REST, 329
RIA, Rich Internet Application, 267
RMI, Remote Method Invocation, 609
rodzaje
    metadanych, 665
    testów, 457
    ziaren EJB, 613
rozdzielacze, 726
rozdzielanie operacji, 726
rozszerzanie klas DAO, 565
rozszerzenie adresu URL, 291
równoległość, 775
równoważenie obciążenia, 787
rzutowanie danych, 349

S

scalanie kolekcji, 33
schemat util, 47
screen scraping, 730
SEDA, Staged Event-Driven Architecture, 739
serializacja, marshalling, 610
serializowanie
    dokumentów XML, 648–653
    obiektów, 609

Kup ksi k   Pole ksi k

serwer
  ActiveMQ, 707
  Apache James Server, 673
  Apache Pluto, 501, 503
  Apache Tomcat, 409, 627
  dm Server, 832
  EAI, 711
  Grizzly, 388
  JBoss, 804
  LDAP, 182
  MBeans, 663
  OpenDS, 182
  SMTP, 676
serwery transakcyjne, 747
serwlet, 208
  DispatcherServlet, 273, 277, 374
  DwrServlet, 224
  FacesServlet, 256
sesje kontekstowe platformy Hibernate, 567
setter instrukcji, 528
settery, 40, 92
SID, Security Identity, 192
silnik bazodanowy Apache Derby, 250
skalowalność, 752, 775
skanowanie
  adnotacji, 273
  komponentów, 65, 67
składnia języka, 78
skracanie konfiguracji, 88
skróty, 28
skrypt, 153
  activemq.sh, 692
  gridgain.sh, 786
skrypty z ziarnami, 154, 156
słowo kluczowe
  and, 142
  newsfeed, 337
SMTP, 673
specyfikacja
  JSR-220, 551
  SQL-99, 577
sprawdzanie
  nagłówków, 292
  poprawności
    danych, 307, 316, 515
    działania, 463
    obiektu, 416
    stanu, 463
    ziaren, 319
  właściwości, 49
    typów obiektowych, 50
    typów prostych, 50
    wszystkich typów, 50
    z adnotacją @Required, 51
  zależności, 49, 51, 56
Spring, *Patrz* platforma
standard
  AMQP, 687
  JAX-RPC, 631
  JAX-WS, 631
  JSR-303, 319, 320
  BPM, 800
stany, 138, 230, 236
  akcji, 242
  końcowe, 244
  obiektu, 777
  podejmowania decyzji, 243
  podprzepływów, 246
  SQL-a, 544
  widoku, 239
statyczne metody fabryczne, 71
sterownik JDBC, 420
struktura katalogów Grails, 405
STS, SpringSource Tool Suite, 234, 834
style integracji, 711
sygnatury
  metod, 132
  typów, 134
symbol wieloznaczny, 767
synchroniczne odbieranie komunikatów, 701
system
  CRM, 448, 454
  jBPM, 799, 802
  jBPM 4, 804
  pamięci trwałej, 420
  plików, 718
szablon
  e-maili, 676
  JmsTemplate, 707
  ponawiania prób, 763
  TransactionTemplate, 585
szablony
  Hibernate, 562
  JDBC, 487, 526, 530, 537, 540
  JMS, 693, 696
  JPA, 562
  ORM Springa, 562
  transakcji, 585
szyfrowanie haseł, 180

Ś

ścieżki do zasobów, 95
środowisko
  Adobe Flex Builder, 358
  Eclipse, 834
  Equinox, 819
  Flex Builder, 362
  IDE, 452
  IDEA 9.0, 363

Kup książkę Poleć książkę

środowisko
  produkcyjne, 410
  rozwojowe, 410
  Spring IDE, 834
  SpringSource Tool Suite, 440
  STS, 445
  testowe, 410
  uruchomieniowe OSGi, 826

T

tabela BOOK_STOCK, 577
tablica, 29
technologia
  BlazeDS, 374, 378
  GORM, 432
  Hessian, 626
  Hibernate, 548
  JDBC, 519
  JMS, 687
  JMX, 657
  JSF, 255
  OSGi, 815
  OXM, 649
  REST, 329
  RMI, 609
testowanie
  klas niezależnych, 463
  klas zależnych, 466
  poziomów izolacji, 598
  propagacji, 593
testy, 457
  integracyjne, 424, 463, 470, 478
    dostęp do bazy, 487
    zarządzanie transakcjami, 482
  jednostkowe, 424, 428, 463
  jednostkowe kontrolerów, 471
  w JUnit 4, 460
  w TestNG, 461
tokeny statyczne, 176
transakcja, 576, 761
  atomowość, 576
  izolacja, 576
  spójność, 576
  trwałość, 576
transakcje
  JMS, 700
  współbieżne, 596, 597
tryb
  aspectj, 608
  push HTTP, 387
  REQUIRED, 593
  REQUIRES_NEW, 595
tryby
  automatycznego łączenia ziaren, 53
  portletu, 505
tworzenie
  aplikacji, 203
    Fleksa, 360, 362
    Grails, 404, 406
    sieciowych, 213, 267
    Spring-WS, 641
  definicji przepływów sterowania, 234
  edytorów właściwości, 108
  egzemplarza
    kontekstu aplikacji, 25
    kontenera IoC, 23
  e-maili, 676
  frontonu, 452
  klas domenowych, 269, 412
  klienta, 645
  komponentów, 400
    EJB 2.x, 613, 614, 616
    EJB 3.0, 621
  komunikatów, 637, 643
  kontrolerów, 167, 274
    formularzy, 300, 512
    formularzy kreatora, 314
    portletów, 499
  nazw plików, 325
  niestandardowej klasy, 306
  niestandardowych znaczników, 434
  obiektów
    MDP, 713
    POJO, 701
  odwzorowań ORM, 547, 548, 562
  plików konfiguracyjnych, 27, 166, 232, 272, 498
  portletów, 495
  postprocesorów ziaren, 96
  procesu biznesowego, 812
  przepływów JSF, 258
  pul wątków, 110
  punktów końcowych, 649
  punktów końcowych usług, 642, 653
  rusztowania, 409
  stron formularza kreatora, 311
  szablonów JDBC, 535
  testów, 458, 463
  testów integracyjnych, 470, 472
  usług, 809
    sieciowych, 648
    sieciowych SOAP, 628
  węzła gridu, 792
  widoków, 235, 341
    formularzy, 298
    JSF, 256
    JSP, 277
    PDF, 324
    portletów, 501
    stron, 167
    w Excelu, 323
  ziaren, 23, 26, 34, 45, 71, 72
  ziaren EJB, 618

Kup ksi k   Pole ksi k

typy
  mediów, 293
  niestandardowe, 305
  stanów, 237
  układów, 424

U

udostępnianie
  usług, 378, 625
    mechanizm HTTP Invoker, 628
    Burlapa, 627
    RMI, 611
    platforma CXF, 633
    punkty końcowe JAX-WS, 631
    sieciowych SOAP, 630
  wstępnych danych, 303
  ziaren, 225, 664
ukrycie obsługi komunikatów, 742
umiędzynarodawianie komunikatów, 417
unikanie powtórzeń, 400
uprawnienia, 192, 198
uruchamianie
  aplikacji Grails, 406
  obiektów DAO, 525
  zadań, 769
    na podstawie planu, 771
    w aplikacji sieciowej, 770
    z parametrami, 773
    z wiersza poleceń, 770
usługa
  helloworld-service, 816
  logowania, 164, 173
  przetwarzająca formularz, 300
  tworzenia przepływów JSF, 257, 258
  w pakunku, 819
  wylogowywania, 164, 175
usługi
  AMF, 372, 378
  Burlapa, 627
  Hessiana, 627
  JMS, 387, 390
  oparte na komunikatach, 387
  REST, 329
    biblioteka Project Rome, 356
    dostęp do usług, 333
    format JSON, 345
    format XML, 348
    publikowanie danych aplikacji, 329, 330
  RMI, 611
  sieciowe, 609, 610
    z kontraktem na końcu, 629
    z kontraktem na początku, 629
  sieciowe SOAP, 370, 628
  typu REST, 333, 345
  zgodne z interfejsami, 830
ustawianie nagłówków HTTP, 326
ustawienia
  JDBC, 520
  regionalne, 285, 287
    atrybut sesji, 286
    nagłówek żądania HTTP, 286
    plik cookie, 286
    zmienianie, 286
usuwanie ziaren, 84, 87
utrwalanie obiektów, 249
  adnotacje JPA, 551, 553
  interfejs API Hibernate, 551
  odwzorowania ORM, 548, 562
  platforma Hibernate, 553
  sesje kontekstowe Hibernate, 567
  wstrzykiwanie kontekstu JPA, 570
uwierzytelnianie, 163
  HTTP, 164, 173
  użytkowników, 176, 181
użytkownik pluto, 502
używanie
  agregatorów, 728
  bram, 740
  interceptorów przetwarzania, 282
  JDBC, 526
  języka SpEL, 79
  JPA, 252, 548
  komponentów JSF, 261
  nazwanych parametrów, 540
  Quartza, 679, 681
  rozdzielacza, 726
  schematu JMS, 706
  serwera dm Server, 832
  szablonów JDBC, 530

W

walidator, 308, 317
warunkowe
  przekazywanie komunikatów, 729
  wykonywanie kroków, 767, 768
wczytywanie
  danych, 753
  kontekstu aplikacji, 214
  pakietów zasobów, 101
  zasobów, 93
wewnątrzwierszowe definiowanie kodu, 161
węzeł, 785
węzeł gridu, 792
wiadomości na Twitterze, 731
wiadomość MIME, 677
wiązanie
  właściwości, 305
  wyjątków, 294

Kup ksi k Pole ksi k

żądań, 278
na poziomie klasy, 279
na podstawie parametru, 508
na podstawie typu, 281
sieciowych, 642
w metodach, 279
z trybem portletu, 505
widok, 167, 235, 239, 267, 291, 294
formularza, 169
JSF, 256
logiczny, 331, 341
membertemplate, 332
PDF, 324
portletu, 501
wielodziedziczenie, 137
wieloznaczność, 37
wiersz poleceń, 770
właściwości
JDBC, 576
obiektu, 75
typów obiektowych, 50
typów prostych, 50
wszystkich typów, 50
ziaren, 28
właściwość
location, 289
propertyPath, 76
serviceInterface, 612
serviceUrl, 612
transactionManager, 761
wpisy, entries, 344
wplatanie, weaving, 143
wplatanie w czasie ładowania, 146, 605
wprowadzenie, introduction, 119, 136
WSDL, Web Services Description Language, 628
współbieżność, 109, 112, 766
wstrzykiwanie
danych, 51
konfiguracji testów, 478–481
kontekstu JPA, 570
nazwanej zmiennej, 79
szablonu JDBC, 535
zależności, 217, 398
zasobów, 95
ziaren, 149, 157–159
wtyczka
App Engine, 409
Clojure, 409
ContextLoaderPlugin, 214, 216
dm Server Tools, 834
Granite Data Service Gas3, 387
m2eclipse, 446, 447
Quartz, 409
Spring WS, 409
wtyczki platformy Grails, 408
wycofywanie
transakcji, 602
operacji, 762
wygasanie danych, 309
wyjątek
BeanInitializationException, 52
ClassCastException, 44
CreateException, 618
DuplicateKeyException, 544
IllegalArgumentException, 71
JmsException, 695, 703
NamingException, 618
NullPointerException, 86
RemoteException, 615, 618
ReservationNotAvailableException, 300
SQLException, 580
UnsatisfiedDependencyException, 50, 56
wyjątki czasu wykonania, 618
wykrywanie ziaren MBeans, 666
wyodrębnianie treści, 715
wyrażenia
cron, 116, 681
z punktami przecięcia, 132
wysyłanie
e-maili, 657, 672–675
komunikatów, 396, 693, 696
komunikatów JMS, 688
powiadomień, 673
wiadomości MIME, 677
wyszukiwanie
informacji, 716
usług, 828
wyświetlanie
danych, 496
warunkowe, 191
widoku formularza, 514
zawartości widoków, 191
wywoływanie usług, 625
Burlapa, 628
Hessiana, 627
mechanizm HTTP Invoker, 628
RMI, 612
wywoływanie usług sieciowych
platforma CXF, 635
platforma Spring-WS, 645
serializowanie dokumentów XML, 652
SOAP, 630
wywoływanie zdalne, 612
wyzwalacz
CronTrigger, 681
SimpleTrigger, 681
wzorzec, 132, 319
adresu URL, 164
MVP, 357
nazwy ziarna, 134
projektowy DAO, 521

Kup ksi k Pole ksi k

X

XML, 140
- deklarowanie aspektów, 141
- deklarowanie punktów przecięcia, 141
- deklarowanie rad, 142
- deklarowanie wprowadzeń, 142

Z

zabezpieczanie
- metod, 189
- obiektów domenowych, 192
- przepływów sterowania, 247
- wywołań metod, 188

zabezpieczenia w widokach, 190
zagadnienia przecinające, 122
zagnieżdżenie interceptora, 188
zależności między ziarnami, 49
zapamiętywanie użytkowników, 164, 176
zapełnianie formularza, 302
zapisywanie danych, 758
zapytania GORM, 432
zarządzane ziarna JSF, 218
zarządzanie
- kontekstem aplikacji, 472
- listami ACL, 196
- procesami biznesowymi, 798, 812, 814
- przepływem pracy, 801
- przepływem sterowania, 229
- transakcjami, 482–487, 564, 568, 575
  - adnotacja @Transactional, 590
  - deklaratywne, 575, 588, 590
  - implementacja menedżera, 581
  - interfejs menedżera, 583
  - izolacja, 596
  - JMS, 700, 705
  - limit czasu, 604
  - metoda commit(), 580
  - metoda rollback(), 580
  - poziomy izolacji, 597
  - programowe, 575, 583, 585
  - propagacja, 592
  - rady, 588
  - szablon, 585
  - tryb tylko do odczytu, 604
  - wplatanie w czasie ładowania, 605
  - wycofywanie, 602
- ziarnami MBeans, 664

zasada DRY, 400
zasięg ziarna, 82
zasoby, resources, 665
- kontenera, 92
- zewnętrzne, 93

zastosowanie Terracotty, 778
zbiory, 29
zdalna obsługa porcji, 767
zdalne
- wywołania, 225, 609
- wywołania Flasha, 372
- ziarna MBeans, 669, 671

zdarzenia, 102, 103
ziarna, 23
- EJB, 609
- fabryczne, 45, 47, 557
- MBeans, 662, 664
- MBeans modelu, 657
- MBeans technologii JMX, 657
- nadrzędne, 33, 62
- podrzędne, 33, 62
- sesyjne
  - bezstanowe, 613, 616
  - stanowe, 613
  - sterowane komunikatami, 613, 701
- wewnętrzne, 42
- zarządzane, 657
- zarządzane JSF, 223

ziarno
- do negocjowania treści, 292, 293
- documentReplicator, 666
- interceptora, 285
- LocalEntityManagerFactoryBean, 561
- określające widoki, 325
- SchedulerFactoryBean, 682

zmiana serwera HTTP, 832
zmienne środowiskowe, 404
znacznik
- <aop:aspect>, 141
- <beans>, 48
- <constructor-app>, 37
- <context:mbean-export>, 667
- <entry>, 32
- <form:form>, 299
- <form:input>, 302
- <form:select>, 303
- <g:message/>, 419
- <g:render>, 431
- <item>, 354
- <jee:jndi-lookup>, 624
- <key>, 31
- <link>, 354
- <map>, 31
- <mx:Button />, 361
- <prop>, 33
- <props>, 33
- <value>, 44

Kup ksi k Pole ksi k

Ż

żądanie
HTTP GET, 275
HTTP POST, 276, 309, 320
wykonania akcji, 496, 513
wyświetlenia danych, 496, 513

Kup ksi k Pole ksi k

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

http://program-partnerski.helion.pl

GRUPA WYDAWNICZA
Helion SA

Wykorzystaj potencjał Springa i zoptymalizuj swoją pracę!

Spring zadebiutował na rynku w 2004 roku, osiem lat po opublikowaniu pierwszej wersji języka Java — i od tego czasu jest dynamicznie rozwijany. Dzięki licznym modułom pozwala błyskawicznie tworzyć skomplikowane aplikacje i wyręcza programistów w trudzie ustawiania typowych konfiguracji. Jeżeli do tego dołożyć ogromną społeczność i świetną dokumentację, to nie ma się co dziwić, że jest tak popularny!

Jeżeli wykorzystujesz Springa w swojej codziennej pracy lub chcesz wypróbować jego możliwości, trafiłeś na doskonałą książkę. Należy ona do cenionej wśród programistów serii „Receptury". Znajdziesz tu omówienie zarówno podstawowych zagadnień związanych ze Springiem, jak i tych zaawansowanych. Na samym początku poznasz kontener IoC (ang. Inversion of Control), nauczysz się tworzyć ziarna oraz wstrzykiwać je na różne sposoby. W kolejnych rozdziałach odkryjesz, jak korzystać z AspectJ, Spring Web Flow oraz Spring-WS. Ponadto zobaczysz, jak używać REST, testów jednostkowych i integracyjnych oraz ORM. Spring wspiera te i wiele innych obszarów codziennej pracy programisty. Książka ta jest doskonałym omówieniem Springa na podstawie przykładów jego zastosowań.

Dzięki tej książce:

- stworzysz kontener IoC
- skonstruujesz ziarna i wstrzykniesz je
- poznasz dostępne moduły
- przygotujesz usługi sieciowe i skorzystasz z tych usług

helion.pl
księgarnia internetowa

Nr katalogowy: 26480

Księgarnia internetowa:
http://helion.pl

Zamówienia telefoniczne:
0 801 339900
0 601 339900

Apress®

Helion

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Informatyka w najlepszym wydaniu

ISBN 978-83-246-8226-3
9 788324 682263
cena: 119,00 zł