

Wydanie II



Christian Bauer
Gavin King
Gary Gregory

Java Persistence

Programowanie aplikacji bazodanowych w **Hibernate**

Helion 

Tytuł oryginału: Java Persistence with Hibernate, 2nd Edition

Tłumaczenie: Radosław Meryk

Projekt okładki: Studio Gravite / Olsztyn; Obarek, Pokoński, Pazdrijowski, Zaprucki
Materiały graficzne na okładce zostały wykorzystane za zgodą Shutterstock Images LLC.

ISBN: 978-83-283-2782-5

Original edition copyright © 2016 by Manning Publications Co.
All rights reserved.

Polish edition copyright © 2017 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 jakiegokolwiek 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 biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION nie ponoszą 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)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
<http://helion.pl/user/opinie/javpe2>
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Słowo wstępne do pierwszego wydania	17
Przedmowa	19
Podziękowania	21
O książce	23
O autorach	27

CZĘŚĆ I WPROWADZENIE W TEMATYKĘ ORM 29

Rozdział 1. Utrwalanie obiektowo-relacyjne 31

1.1. Co to jest utrwalanie?	32
1.1.1. Relacyjne bazy danych	33
1.1.2. Język SQL	34
1.1.3. Korzystanie z języka SQL w Javie	35
1.2. Niedopasowanie paradygmatów	37
1.2.1. Problem ziarnistości	38
1.2.2. Problem podtypów	40
1.2.3. Problem tożsamości	41
1.2.4. Problemy związane z asocjacjami	43
1.2.5. Problem poruszania się po danych	44
1.3. ORM i JPA	45
1.4. Podsumowanie	47

Rozdział 2. Zaczynamy projekt 49

2.1. Wprowadzenie do frameworka Hibernate	49
2.2. Aplikacja „Witaj, świecie” z JPA	50
2.2.1. Konfigurowanie jednostki utrwalania	51
2.2.2. Pisanie klasy utrwalania	53
2.2.3. Zapisywanie i ładowanie komunikatów	54
2.3. Natywne mechanizmy konfiguracji frameworka Hibernate	57
2.4. Podsumowanie	60

Rozdział 3. Modele dziedziny i metadane 61

3.1. Przykładowa aplikacja CaveatEmptor	62
3.1.1. Architektura warstwowa	62
3.1.2. Analiza dziedziny biznesowej	64
3.1.3. Model dziedziny aplikacji CaveatEmptor	65

- 3.2. Implementacja modelu dziedziny 66
 - 3.2.1. Rozwiązanie problemu wyciekania obszarów zainteresowania 67
 - 3.2.2. Przezroczyste i zautomatyzowane utrwalanie 68
 - 3.2.3. Pisanie klas zdolnych do utrwalania 69
 - 3.2.4. Implementacja asocjacji POJO 73
- 3.3. Metadane modelu dziedziny 77
 - 3.3.1. Metadane bazujące na adnotacjach 78
 - 3.3.2. Stosowanie reguł Bean Validation 80
 - 3.3.3. Pobieranie metadanych z zewnętrznych plików XML 83
 - 3.3.4. Dostęp do metadanych w fazie działania aplikacji 87
- 3.4. Podsumowanie 90

CZĘŚĆ II STRATEGIE MAPOWANIA 91

Rozdział 4. Mapowanie klas utrwalania 93

- 4.1. Encje i typy wartości 93
 - 4.1.1. Droбноziarniste modele dziedziny 94
 - 4.1.2. Definiowanie pojęć aplikacji 94
 - 4.1.3. Odróżnianie encji od typów wartości 96
- 4.2. Mapowanie encji z tożsamością 97
 - 4.2.1. Tożsamość a równość w Javie 98
 - 4.2.2. Pierwsza klasa encji i mapowanie 98
 - 4.2.3. Wybieranie klucza głównego 100
 - 4.2.4. Konfigurowanie generatorów kluczy 101
 - 4.2.5. Strategie generatorów identyfikatorów 104
- 4.3. Opcje mapowania encji 108
 - 4.3.1. Zarządzanie nazwami 108
 - 4.3.2. Dynamiczne generowanie SQL 111
 - 4.3.3. Encje niezmiennie 112
 - 4.3.4. Mapowanie encji na podzapytanie 113
- 4.4. Podsumowanie 114

Rozdział 5. Mapowanie typów wartości 117

- 5.1. Mapowanie prostych właściwości 118
 - 5.1.1. Przesłanie domyślnego zachowania dla właściwości o typach prostych 119
 - 5.1.2. Personalizacja dostępu do właściwości 120
 - 5.1.3. Używanie właściwości wyprowadzonych 122
 - 5.1.4. Transformacje wartości kolumn 123
 - 5.1.5. Wygenerowane i domyślne wartości właściwości 124
 - 5.1.6. Właściwości opisujące czas 125
 - 5.1.7. Mapowanie typów wyliczeniowych 126
- 5.2. Mapowanie komponentów osadzanych 126
 - 5.2.1. Schemat bazy danych 127
 - 5.2.2. Przystosowanie klas do osadzania 127
 - 5.2.3. Przesłanie osadzonych atrybutów 131
 - 5.2.4. Mapowanie zagnieżdżonych komponentów osadzanych 132

- 5.3. Mapowanie typów Java i SQL za pomocą konwerterów 134
 - 5.3.1. *Typy wbudowane* 134
 - 5.3.2. *Tworzenie własnych konwerterów JPA* 140
 - 5.3.3. *Rozszerzanie frameworka Hibernate za pomocą typów użytkownika* 147
- 5.4. Podsumowanie 154

Rozdział 6. Mapowanie dla dziedziczenia 155

- 6.1. Jedna tabela na konkretną klasę. Niejawna obsługa polimorfizmu 156
- 6.2. Jedna tabela na konkretną klasę oraz unie 158
- 6.3. Jedna tabela na hierarchię klas 160
- 6.4. Jedna tabela na podklasę oraz złączenia 164
- 6.5. Mieszane strategie dziedziczenia 167
- 6.6. Dziedziczenie klas osadzanych 169
- 6.7. Wybór strategii 172
- 6.8. Asocjacje polimorficzne 173
 - 6.8.1. *Polimorficzne asocjacje wiele-do-jednego* 173
 - 6.8.2. *Polimorficzne kolekcje* 176
- 6.9. Podsumowanie 177

Rozdział 7. Mapowanie kolekcji i asocjacje pomiędzy encjami 179

- 7.1. Zbiory, kolekcje bag, listy i mapy typów wartości 180
 - 7.1.1. *Schemat bazy danych* 180
 - 7.1.2. *Tworzenie i mapowanie właściwości będących kolekcjami* 180
 - 7.1.3. *Wybór interfejsu kolekcji* 182
 - 7.1.4. *Mapowanie zbioru* 184
 - 7.1.5. *Mapowanie kolekcji bag identyfikatorów* 185
 - 7.1.6. *Mapowanie list* 186
 - 7.1.7. *Mapowanie mapy* 187
 - 7.1.8. *Kolekcje posortowane i uporządkowane* 188
- 7.2. Kolekcje komponentów 191
 - 7.2.1. *Równość egzemplarzy komponentu* 192
 - 7.2.2. *Kolekcja Set komponentów* 194
 - 7.2.3. *Kolekcja bag komponentów* 196
 - 7.2.4. *Mapa wartości komponentów* 197
 - 7.2.5. *Komponenty jako klucze mapy* 198
 - 7.2.6. *Kolekcje w komponencie osadzonym* 199
- 7.3. Mapowanie asocjacji encji 200
 - 7.3.1. *Najprostsza możliwa asocjacja* 201
 - 7.3.2. *Definiowanie asocjacji dwukierunkowych* 202
 - 7.3.3. *Kaskadowe zmiany stanu* 204
- 7.4. Podsumowanie 211

Rozdział 8. Zaawansowane mapowanie asocjacji pomiędzy encjami 213

- 8.1. Asocjacje jeden-do-jednego 214
 - 8.1.1. Współdzielenie klucza głównego 214
 - 8.1.2. Generator kluczy obcych i głównych 217
 - 8.1.3. Wykorzystanie kolumny złączenia klucza obcego 220
 - 8.1.4. Korzystanie z tabeli złączenia 221
- 8.2. Asocjacje jeden-do-wielu 224
 - 8.2.1. Kolekcje bag jeden-do-wielu 224
 - 8.2.2. Jednokierunkowe i dwukierunkowe mapowania list 226
 - 8.2.3. Opcjonalna asocjacja jeden-do-wielu z tabelą złączenia 228
 - 8.2.4. Asocjacje jeden-do-wielu w klasach osadzanych 230
- 8.3. Asocjacje wiele-do-wielu i asocjacje potrójne 232
 - 8.3.1. Jednokierunkowe i dwukierunkowe asocjacje wiele-do-wielu 232
 - 8.3.2. Asocjacja wiele-do-wielu z pośrednią encją 234
 - 8.3.3. Asocjacje trójelementowe z komponentami 238
- 8.4. Asocjacje pomiędzy encjami z wykorzystaniem kolekcji Map 241
 - 8.4.1. Relacje jeden-do-wielu z kluczem w postaci właściwości 241
 - 8.4.2. Relacje trójczłonowe klucz-wartość 243
- 8.5. Podsumowanie 244

Rozdział 9. Schematy złożone i odziedziczone 245

- 9.1. Ulepszanie schematu bazy danych 246
 - 9.1.1. Dodawanie pomocniczych obiektów bazy danych 247
 - 9.1.2. Ograniczenia SQL 250
 - 9.1.3. Tworzenie indeksów 257
- 9.2. Obsługa kluczy odziedziczonych 258
 - 9.2.1. Mapowanie naturalnych kluczy głównych 258
 - 9.2.2. Mapowanie złożonych kluczy głównych 259
 - 9.2.3. Klucze obce w złożonych kluczach głównych 261
 - 9.2.4. Klucze obce odwołujące się do złożonych kluczy głównych 264
 - 9.2.5. Klucze obce odwołujące się do pól niebędących kluczami głównymi 265
- 9.3. Mapowanie właściwości do tabel pomocniczych 266
- 9.4. Podsumowanie 268

CZĘŚĆ III TRANSAKCYJNE PRZETWARZANIE DANYCH 269**Rozdział 10. Zarządzanie danymi 271**

- 10.1. Cykl życia utrwalania 272
 - 10.1.1. Stany egzemplarza encji 273
 - 10.1.2. Kontekst utrwalania 274
- 10.2. Interfejs EntityManager 276
 - 10.2.1. Kanoniczna jednostka pracy 276
 - 10.2.2. Utrwalanie danych 278
 - 10.2.3. Pobieranie i modyfikowanie trwałych danych 279
 - 10.2.4. Pobieranie referencji 281
 - 10.2.5. Przelączanie danych do stanu przejściowego 282
 - 10.2.6. Odświeżanie danych 284

- 10.2.7. *Replikowanie danych* 284
- 10.2.8. *Buforowanie w kontekście utrwalania* 285
- 10.2.9. *Synchronizowanie kontekstu utrwalania* 287
- 10.3. *Praca z encjami w stanie odłączonym* 288
 - 10.3.1. *Tożsamość odłączonych egzemplarzy* 289
 - 10.3.2. *Implementacja metody równości* 291
 - 10.3.3. *Odlączanie egzemplarzy encji* 294
 - 10.3.4. *Scalanie egzemplarzy encji* 295
- 10.4. *Podsumowanie* 297

Rozdział 11. Transakcje i współbieżność 299

- 11.1. *Podstawowe wiadomości o transakcjach* 300
 - 11.1.1. *Cechy ACID* 300
 - 11.1.2. *Transakcje bazodanowe i systemowe* 301
 - 11.1.3. *Transakcje programowe z JTA* 301
 - 11.1.4. *Obsługa wyjątków* 303
 - 11.1.5. *Deklaratywne rozgraniczanie transakcji* 306
- 11.2. *Zarządzanie współbieżnym dostępem* 306
 - 11.2.1. *Współbieżność na poziomie bazy danych* 307
 - 11.2.2. *Optymistyczne zarządzanie współbieżnością* 312
 - 11.2.3. *Jawne pesymistyczne blokady* 320
 - 11.2.4. *Unikanie zakleszczeń* 324
- 11.3. *Dostęp do danych na zewnątrz transakcji* 325
 - 11.3.1. *Czytanie danych w trybie autozatwierdzenia* 326
 - 11.3.2. *Kolejkowanie modyfikacji* 328
- 11.4. *Podsumowanie* 329

Rozdział 12. Plany pobierania, strategie i profile 331

- 12.1. *Ładowanie leniwe i zachłanne* 332
 - 12.1.1. *Obiekty proxy encji* 333
 - 12.1.2. *Leniwe ładowanie trwałych kolekcji* 337
 - 12.1.3. *Leniwe ładowanie z przechwytywaniem* 339
 - 12.1.4. *Zachłanne ładowanie asocjacji i kolekcji* 342
- 12.2. *Wybór strategii pobierania* 344
 - 12.2.1. *Problem $n+1$ instrukcji SELECT* 345
 - 12.2.2. *Problem iloczynu kartezjańskiego* 346
 - 12.2.3. *Pobieranie danych partiami z wyprzedzeniem* 349
 - 12.2.4. *Pobieranie kolekcji z wyprzedzeniem z wykorzystaniem podzapytań* 351
 - 12.2.5. *Pobieranie zachłanne z wieloma instrukcjami SELECT* 352
 - 12.2.6. *Dynamiczne pobieranie zachłanne* 353
- 12.3. *Korzystanie z profili pobierania* 355
 - 12.3.1. *Deklarowanie profili pobierania Hibernate* 356
 - 12.3.2. *Korzystanie z grafów encji* 357
- 12.4. *Podsumowanie* 361

Rozdział 13. Filtrowanie danych 363

- 13.1. Kaskadowe przejścia stanu 364
 - 13.1.1. Dostępne opcje kaskadowe 365
 - 13.1.2. Przechodnie odłączanie i scalanie 366
 - 13.1.3. Kaskadowe odświeżanie 368
 - 13.1.4. Kaskadowe replikacje 370
 - 13.1.5. Włączanie globalnej opcji przechodniego utrwalania 371
- 13.2. Nasłuchiwanie i przechwytywanie zdarzeń 372
 - 13.2.1. Obserwatory zdarzeń i wywołania zwrotne JPA 372
 - 13.2.2. Implementacja interceptorów Hibernate 376
 - 13.2.3. Rdzeń systemu obsługi zdarzeń 380
- 13.3. Audyt i wersjonowanie z wykorzystaniem Hibernate Envers 382
 - 13.3.1. Aktywacja rejestrowania audytu 382
 - 13.3.2. Tworzenie śladu audytu 384
 - 13.3.3. Szukanie wersji 385
 - 13.3.4. Dostęp do historycznych danych 386
- 13.4. Dynamiczne filtry danych 389
 - 13.4.1. Definiowanie dynamicznych filtrów 390
 - 13.4.2. Stosowanie filtra 390
 - 13.4.3. Włączanie filtra 391
 - 13.4.4. Filtrowanie dostępu do kolekcji 392
- 13.5. Podsumowanie 393

CZĘŚĆ IV PISANIE ZAPYTAŃ 395**Rozdział 14. Tworzenie i uruchamianie zapytań 397**

- 14.1. Tworzenie zapytań 398
 - 14.1.1. Interfejsy zapytań JPA 398
 - 14.1.2. Wyniki zapytania z określonym typem danych 401
 - 14.1.3. Interfejsy zapytań frameworka Hibernate 402
- 14.2. Przygotowywanie zapytań 403
 - 14.2.1. Zabezpieczenie przed atakami wstrzykiwania SQL 403
 - 14.2.2. Wiązanie parametrów nazwanych 404
 - 14.2.3. Korzystanie z parametrów pozycyjnych 406
 - 14.2.4. Stronicowanie dużych zbiorów wyników 406
- 14.3. Uruchamianie zapytań 408
 - 14.3.1. Wyszczególnianie wszystkich wyników 408
 - 14.3.2. Pobieranie pojedynczego wyniku 408
 - 14.3.3. Przewijanie z wykorzystaniem cursorów bazy danych 409
 - 14.3.4. Iterowanie po wynikach 411
- 14.4. Nazywanie i eksternalizacja zapytań 412
 - 14.4.1. Wywoływanie zapytania przez nazwę 412
 - 14.4.2. Definiowanie zapytań w metadanych XML 413
 - 14.4.3. Definiowanie zapytań z adnotacjami 414
 - 14.4.4. Programowe definiowanie nazwanych zapytań 415
- 14.5. Podpowiedzi do zapytań 416
 - 14.5.1. Ustawianie limitu czasu 417
 - 14.5.2. Ustawianie trybu synchronizacji 417

- 14.5.3. Ustawianie trybu tylko do odczytu 418
- 14.5.4. Ustawianie rozmiaru pobierania 418
- 14.5.5. Ustawianie komentarza SQL 419
- 14.5.6. Podpowiedzi nazwanych zapytań 419
- 14.6. Podsumowanie 421

Rozdział 15. Języki zapytań 423

- 15.1. Selekcja 424
 - 15.1.1. Przypisywanie aliasów i rdzeni zapytań 425
 - 15.1.2. Zapytania polimorficzne 426
- 15.2. Ograniczenia 427
 - 15.2.1. Wyrażenia porównań 428
 - 15.2.2. Wyrażenia z kolekcjami 432
 - 15.2.3. Wywoływanie funkcji 433
 - 15.2.4. Sortowanie wyników zapytania 436
- 15.3. Rzutowanie 437
 - 15.3.1. Rzutowanie encji i wartości skalarnych 437
 - 15.3.2. Dynamiczne tworzenie egzemplarzy 439
 - 15.3.3. Niepowtarzalność wyników 440
 - 15.3.4. Wywoływanie funkcji w projekcjach 441
 - 15.3.5. Funkcje agregacji 443
 - 15.3.6. Grupowanie 445
- 15.4. Złączenia 446
 - 15.4.1. Złączenia w SQL 447
 - 15.4.2. Opcje złączenia w JPA 449
 - 15.4.3. Niejawne złączenia asocjacyjne 449
 - 15.4.4. Złączenia jawne 451
 - 15.4.5. Dynamiczne pobieranie ze złączeniami 453
 - 15.4.6. Złączenia teta 456
 - 15.4.7. Porównywanie identyfikatorów 457
- 15.5. Podzapytania 459
 - 15.5.1. Zagnieżdżanie skorelowane i nieskorelowane 460
 - 15.5.2. Kwantyfikacja 461
- 15.6. Podsumowanie 462

Rozdział 16. Zaawansowane opcje zapytań 465

- 16.1. Transformacje wyników zapytań 466
 - 16.1.1. Zwracanie listy list 466
 - 16.1.2. Zwracanie listy map 467
 - 16.1.3. Mapowanie aliasów na właściwości JavaBean 468
 - 16.1.4. Pisanie własnej implementacji klasy ResultTransformer 469
- 16.2. Filtrowanie kolekcji 470
- 16.3. API Criteria frameworka Hibernate 473
 - 16.3.1. Selekcja i porządkowanie 473
 - 16.3.2. Ograniczanie 474
 - 16.3.3. Rzutowanie i agregacja 475

- 16.3.4. *Złączenia* 477
- 16.3.5. *Podzapytania* 478
- 16.3.6. *Zapytania przez przykład* 479
- 16.4. *Podsumowanie* 482

Rozdział 17. Dostosowywanie SQL 483

- 17.1. *Korzystanie z trybu JDBC* 484
- 17.2. *Mapowanie wyników zapytania SQL* 486
 - 17.2.1. *Rzutowanie z zapytaniami SQL* 487
 - 17.2.2. *Mapowanie na klasy encji* 488
 - 17.2.3. *Dostosowywanie mapowania wyników* 490
 - 17.2.4. *Eksternalizacja natywnych zapytań* 502
- 17.3. *Dostosowywanie operacji CRUD* 505
 - 17.3.1. *Własne mechanizmy ładujące* 505
 - 17.3.2. *Personalizacja operacji tworzenia, aktualizowania i usuwania egzemplarzy encji* 507
 - 17.3.3. *Personalizacja operacji na kolekcjach* 509
 - 17.3.4. *Zachłanne pobieranie we własnych mechanizmach ładujących* 511
- 17.4. *Wywoływanie procedur składowanych* 513
 - 17.4.1. *Zwracanie zbioru wyników* 515
 - 17.4.2. *Zwracanie wielu zbiorów wyników oraz liczników aktualizacji* 516
 - 17.4.3. *Ustawianie parametrów wejściowych i wyjściowych* 518
 - 17.4.4. *Zwracanie kursora* 521
- 17.5. *Wykorzystywanie procedur składowanych do operacji CRUD* 522
 - 17.5.1. *Spersonalizowany mechanizm ładujący z procedurą* 523
 - 17.5.2. *Procedury dla operacji CUD* 524
- 17.6. *Podsumowanie* 526

CZĘŚĆ V BUDOWANIE APLIKACJI 527

Rozdział 18. Projektowanie aplikacji klient-serwer 529

- 18.1. *Tworzenie warstwy utrwalania* 530
 - 18.1.1. *Generyczny wzorzec obiektu dostępu do danych* 532
 - 18.1.2. *Implementacja generycznego interfejsu* 533
 - 18.1.3. *Implementacja obiektów DAO encji* 536
 - 18.1.4. *Testowanie warstwy utrwalania* 538
- 18.2. *Budowa serwera bezstanowego* 540
 - 18.2.1. *Edycja przedmiotu aukcji* 540
 - 18.2.2. *Składanie oferty* 543
 - 18.2.3. *Analiza aplikacji bezstanowej* 546
- 18.3. *Budowanie serwera stateful* 548
 - 18.3.1. *Edycja przedmiotu aukcji* 549
 - 18.3.2. *Analiza aplikacji stateful* 554
- 18.4. *Podsumowanie* 556

Rozdział 19. Budowanie aplikacji webowych 557

- 19.1. Integracja JPA z CDI 558
 - 19.1.1. Tworzenie obiektu *EntityManager* 558
 - 19.1.2. Łączenie obiektu *EntityManager* z transakcjami 560
 - 19.1.3. Wstrzykiwanie obiektu *EntityManager* 560
- 19.2. Stronicowanie i sortowanie danych 562
 - 19.2.1. Stronicowanie na bazie przesunięć a stronicowanie przez przeszukiwanie 563
 - 19.2.2. Stronicowanie w warstwie utrwalania 565
 - 19.2.3. Odpytywanie strona po stronie 571
- 19.3. Budowanie aplikacji JSF 572
 - 19.3.1. Usługi o zasięgu żądania 572
 - 19.3.2. Usługi o zasięgu konwersacji 576
- 19.4. Serializacja danych modelu dziedziny 585
 - 19.4.1. Pisanie usługi JAX-RS 585
 - 19.4.2. Stosowanie mapowań JAXB 587
 - 19.4.3. Serializacja obiektów proxy frameworka *Hibernate* 589
- 19.5. Podsumowanie 593

Rozdział 20. Skalowanie Hibernate 595

- 20.1. Przetwarzanie masowe i wsadowe 596
 - 20.1.1. Instrukcje masowe w JPQL i API kryteriów 596
 - 20.1.2. Masowe instrukcje w SQL 601
 - 20.1.3. Przetwarzanie wsadowe 602
 - 20.1.4. Interfejs *StatelessSession* frameworka *Hibernate* 606
- 20.2. Buforowanie danych 608
 - 20.2.1. Architektura współdzielonej pamięci podręcznej frameworka *Hibernate* 609
 - 20.2.2. Konfigurowanie współdzielonej pamięci podręcznej 614
 - 20.2.3. Buforowanie encji i kolekcji 616
 - 20.2.4. Testowanie współdzielonej pamięci podręcznej 619
 - 20.2.5. Ustawianie trybów pamięci podręcznej 622
 - 20.2.6. Zarządzanie współdzieloną pamięcią podręczną 623
 - 20.2.7. Pamięć podręczna wyników zapytania 624
- 20.3. Podsumowanie 627

Bibliografia 629**Skorowidz 631**

4

Mapowanie klas utrwalania

W tym rozdziale:

- omówienie encji i typów wartości;
- mapowanie klas encji z tożsamością;
- zarządzanie opcjami mapowania na poziomie encji.

W tym rozdziale zaprezentujemy podstawowe opcje mapowania i wyjaśnimy sposób mapowania klas encji na tabele SQL. Pokażemy i omówimy sposoby obsługi identyfikatorów baz danych i kluczy głównych oraz użycia różnych innych ustawień metadanych pozwalających na spersonalizowanie tego, jak framework Hibernate ładuje i magazynuje egzemplarze klas modelu dziedziny. We wszystkich przykładach mapowania wykorzystamy adnotacje JPA. Najpierw jednak zdefiniujemy podstawowe rozgraniczenie pomiędzy encjami a typami wartości i wyjaśnimy, w jaki sposób powinniśmy podejść do mapowania obiektowo-relacyjnego modelu dziedziny.

Najważniejsza nowa własność w JPA 2

Za pomocą elementu `<delimited-identifiers>` w pliku konfiguracyjnym *persistence.xml* można włączyć globalne „unieszkodliwianie” wszystkich nazw w generowanych instrukcjach SQL.

4.1. Encje i typy wartości

Analizując model dziedziny, można zauważyć różnice pomiędzy klasami: niektóre typy wydają się ważniejsze od innych — reprezentują obiekty biznesowe pierwszej klasy (termin *obiekt* został tutaj użyty w jego podstawowym znaczeniu). Przykładem są klasy

Item, Category i User: są to encje świata rzeczywistego, dla których próbujemy stworzyć reprezentacje (wróć do rysunku 3.3, na którym pokazano przykładowy model dziedziny). Inne typy występujące w modelu dziedziny, takie jak Address, String i Integer, wydają się mniej ważne. W tym podrozdziale opowiemy, co to znaczy używać drobnoziarnistych modeli dziedziny oraz w jaki sposób odróżnić encje od typów wartości.

4.1.1. Drobnoziarniste modele dziedziny

Głównym celem frameworka Hibernate jest wsparcie dla drobnoziarnistych i bogatych modeli dziedziny. Jest to jeden z powodów, dla których korzystamy z obiektów POJO. W uproszczeniu określenie *drobnoziarnisty model* oznacza model, w którym jest więcej klas niż tabel.

Na przykład użytkownik w modelu dziedziny może mieć adres domowy. W bazie danych może występować pojedyncza tabela USERS z kolumnami HOME_STREET, HOME_CITY i HOME_ZIPCODE (pamiętasz problem typów SQL, który omawialiśmy w punkcie 1.2.1?).

W modelu dziedziny moglibyśmy użyć tego samego podejścia — zaprezentować adres w postaci trzech właściwości tekstowych klasy User. Jednak znacznie lepsze jest zamodelowanie adresu z wykorzystaniem klasy Address, gdzie w klasie User występuje właściwość homeAddress. Taki model dziedziny gwarantuje ulepszoną spójność i łatwiejsze ponowne użycie kodu. Ponadto jest bardziej zrozumiały niż SQL z nieelastycznymi systemami typów.

Specyfikacja JPA podkreśla przydatność drobnoziarnistych klas do implementacji bezpieczeństwa typów i zachowań. Wiele osób na przykład modeluje adres e-mail jako tekstową właściwość klasy User. Bardziej zaawansowanym podejściem jest zdefiniowanie klasy EmailAddress, która dodaje bardziej wysokopoziomą semantykę i zachowanie — klasa ta może dostarczać metodę prepareMail(), jednak nie powinna zawierać metody sendMail(), ponieważ nie chcemy, aby klasy modelu dziedziny zależały od podsystemu pocztowego.

Problem ziarnistości prowadzi do rozróżnienia pomiędzy klasami, które w kontekście mechanizmów ORM ma centralne znaczenie. W Javie wszystkie klasy są równe — wszystkie egzemplarze mają własny identyfikator i cykl życia. Gdy wprowadzimy utrwalanie, to może się zdarzyć, że niektóre egzemplarze nie będą mieć własnych identyfikatorów i cyklu życia, lecz będą zależeć od innych. Spróbujmy przeanalizować przykład.

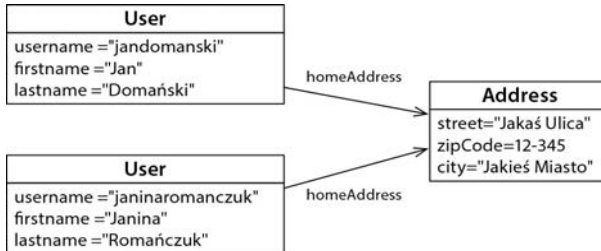
4.1.2. Definiowanie pojęć aplikacji

Dwie osoby mieszkają w tym samym domu i obie mają zarejestrowane konta użytkownika systemu CaveatEmptor. Niechaj będą to Jan i Janina.

Każde konto reprezentuje egzemplarz klasy User. Ponieważ chcemy ładować, zapisywać i usuwać egzemplarze klasy User niezależnie do siebie, to User jest klasą encji, a nie typem wartości. Wyszukiwanie klas encji jest łatwe.

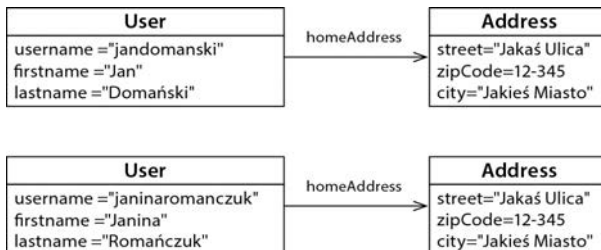
Klasa User ma właściwość homeAddress. Jest to asocjacja z klasą Address. Czy oba egzemplarze klasy User mają w fazie działania aplikacji referencje do tego samego egzemplarza klasy Address, czy też każdy egzemplarz klasy User ma referencję do własnego egzemplarza klasy Address? Czy ma znaczenie to, że Jan i Janina mieszkają w tym samym domu?

Na rysunku 4.1 pokazano sytuację, w której dwa egzemplarze klasy `User` współdzielą egzemplarz klasy `Address` (to jest diagram obiektów UML, a nie diagram klas). Jeżeli obiekt klasy `Address` ma wspierać współdzielenie referencji w fazie działania, to jest to typ encji. Egzemplarz klasy `Address` ma własny cykl życia. Nie można go usunąć, gdy Jan usunie swoje konto `User`. Janina w dalszym ciągu będzie korzystać z referencji do tego egzemplarza klasy `Address`.



Rysunek 4.1. Dwa egzemplarze klasy `User` mają referencje do jednego egzemplarza klasy `Address`

Przyjrzyjmy się teraz modelowi alternatywnemu, w którym każdy obiekt klasy `User` dysponuje referencją do własnego egzemplarza `homeAddress` — tak jak pokazano na rysunku 4.2. W tym przypadku można uzależnić egzemplarz klasy `Address` od egzemplarza klasy `User`: zdefiniujemy go jako typ wartości. Gdy Jan usunie swoje konto `User`, można bezpiecznie usunąć związany z nim egzemplarz klasy `Address`. Ta referencja nie będzie występować w egzemplarzu żadnej innej klasy.



Rysunek 4.2. Dwa egzemplarze klasy `User`, z których każdy ma własny, zależny egzemplarz klasy `Address`

Wprowadzamy zatem następujące, zasadnicze rozróżnienie:

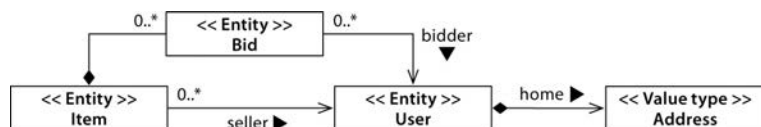
- Możemy pobrać egzemplarz *typu encji*, używając jego tożsamości utrwalania: na przykład egzemplarz `User`, `Item` albo `Category`. Referencja do egzemplarza encji (wskaźnika w JVM) jest utrwalana jako referencja w bazie danych (wartość z ograniczeniem klucza obcego). Egzemplarz encji ma własny cykl życia. Może istnieć niezależnie od dowolnej innej encji. Wybrane klasy modelu dziedziny mapujemy jako typy encji.
- Egzemplarz *typu wartości* nie ma właściwości identyfikatora utrwalania. Należy do egzemplarza encji. Jego cykl życia jest powiązany z egzemplarzem encji, który jest jego właścicielem. Egzemplarz typu wartości nie wspiera współdzielonych referencji. Najbardziej oczywistymi typami wartości są wszystkie klasy zdefiniowane w JDK, takie jak `String`, `Integer`, a nawet typy proste. Można także mapować jako typy wartości klasy należące do naszego własnego modelu dziedziny: na przykład `Address` i `MonetaryAmount`.

W specyfikacji JPA można znaleźć takie samo pojęcie. Jednak typy wartości w specyfikacji JPA są nazywane *prostymi typami właściwości* albo *klasami osadzonymi* (*embeddable classes*). Do tej tematyki powrócimy w następnym rozdziale. Najpierw skoncentrujemy się na encjach.

Identyfikowanie encji i typów wartości w modelu dziedziny nie jest zadaniem realizowanym *ad hoc*, ale lecz zgodnie z konkretną procedurą.

4.1.3. Odróżnianie encji od typów wartości

Do diagramów klas UML można dodać informację na temat stereotypu (mechanizm rozszerzeń UML). Dzięki niemu możliwe jest natychmiastowe rozpoznanie encji i typów wartości. Ta praktyka zmusza także, aby myśleć o zastosowaniu tego rozróżnienia dla wszystkich klas. Jest to pierwszy krok na drodze do optymalnego mapowania oraz wydajnie działającej warstwy utrwalania. Przykład zaprezentowano na rysunku 4.3.



Rysunek 4.3.
Stereotypy dla encji i typów wartości na diagramach UML

Klasy Item i User są oczywistymi encjami. Każda ma własną tożsamość, ich egzemplarze zawierają referencje z wielu innych egzemplarzy (referencje współdzielone) i mają niezależne cykle życia.

Oznaczenie klasy Address jako typu wartości również jest łatwe: pojedynczy egzemplarz klasy User odwołuje się do konkretnego egzemplarza klasy Address. Wiemy to, ponieważ asocjację utworzono jako kompozycję, przy czym egzemplarzowi klasy User została powierzona pełna odpowiedzialność za cykl życia egzemplarza klasy Address. Z tego powodu do egzemplarzy klasy Address nie mogą odwoływać się inne obiekty. Egzemplarze te nie potrzebują też własnej tożsamości.

Problem może stworzyć klasa Bid. W modelowaniu obiektowym jest ona oznaczana jako kompozycja (asocjacja pomiędzy egzemplarzem klasy Item i Bid oznaczona rombem). Egzemplarz Item jest zatem właścicielem jego egzemplarzy klasy Bid i zawiera zbiór referencji. Na pierwszy rzut oka wydaje się to sensowne, ponieważ oferty w systemie aukcji są bezużyteczne w przypadku, gdy towar, na który zostały złożone, jest niedostępny.

Co jednak zrobić, jeśli rozszerzenie modelu dziedziny w przyszłości będzie wymagać kolekcji User#bids, zawierającej wszystkie oferty złożone przez konkretnego użytkownika? W tym momencie asocjacja pomiędzy egzemplarzami Bid i User jest jednokierunkowa; egzemplarz klasy Bid zawiera referencję bidder. Co by było, gdyby była ona dwukierunkowa?

W tym przypadku bylibyśmy zmuszeni do obsługi możliwego współdzielenia referencji z egzemplarzami klasy Bid, więc klasa Bid musiałaby być encją. Ma zależny cykl życia, ale musiałaby mieć też własną tożsamość w celu wsparcia dla (przyszłych) współdzielonych referencji.

Z takim mieszanym zachowaniem można się często spotkać. W takiej sytuacji pierwszą reakcją powinno być przekształcenie wszystkich tego rodzaju klas na typy wartości i promowanie ich do encji tylko wtedy, kiedy to jest absolutnie konieczne. Należy dążyć do upraszczania asocjacji: na przykład utrwalane kolekcje często dodają złożoność bez jakichkolwiek korzyści. Zamiast mapować kolekcje `Item#bids` i `User#bids`, możemy napisać zapytania, aby uzyskać wszystkie oferty dla egzemplarza klasy `Item` oraz te, które złożył określony użytkownik. Asocjacje na diagramie UML będą przebiegały od egzemplarza `Bid` do egzemplarzy `Item` i `User` dokładnie w tym kierunku, a nie w kierunku przeciwnym. Dla klasy `Bid` zostanie użyty stereotyp `<<Value type>>`. Do tego tematu wrócimy w rozdziale 7.

Następnie na podstawie diagramu modelu dziedziny zaimplementujemy obiekty POJO dla wszystkich encji i typów wartości. Należy zatroszczyć się o trzy rzeczy:

- *Współdzielone referencje* — podczas pisania klas POJO należy unikać współdzielenia referencji do egzemplarzy typów wartości. Należy na przykład zadbać o to, aby tylko jeden egzemplarz klasy `User` mógł odwoływać się do egzemplarza klasy `Address`. Klasę `Address` można zaimplementować jako niezmienną, nie definiując publicznej metody `setUser()` i wymuszając relację za pomocą publicznego konstruktora pobierającego argument typu `User`. Oczywiście — zgodnie z tym, o czym mówiliśmy w poprzednim rozdziale — nadal potrzebujemy konstruktora bez argumentów (prawdopodobnie `protected`), tak aby framework Hibernate również mógł tworzyć egzemplarze.
- *Zależności cyklu życia* — jeżeli egzemplarz klasy `User` zostanie usunięty, jego zależność (egzemplarz klasy `Address`) także musi zostać usunięta. Metadane utrwalania zawierają kaskadowe reguły dla wszystkich takich zależności, więc Hibernate (lub baza danych) może zadbać o usunięcie przestarzałych egzemplarzy klasy `Address`. Należy zaprojektować procedury aplikacji i interfejs użytkownika tak, by respektowały takie zależności i ich oczekiwały — trzeba odpowiednio napisać obiekty POJO modelu dziedziny.
- *Tożsamość* — klasy encji wymagają właściwości identyfikatora w prawie wszystkich przypadkach. Klasy typu wartości (i oczywiście klasy JDK takie jak `String` i `Integer`) nie mają właściwości identyfikatora, ponieważ egzemplarze są identyfikowane za pośrednictwem encji właściciela.

Do referencji, asocjacji i reguł cyklu życia powrócimy przy okazji omawiania bardziej zaawansowanych odwzorowań w dalszych rozdziałach niniejszej książki. Następnym tematem będzie tożsamość i właściwości identyfikatorów.

4.2. Mapowanie encji z tożsamością

Mapowanie encji z tożsamością wymaga zrozumienia pojęć tożsamości i równości w Javie. Dopiero po omówieniu tych pojęć przejdziemy do przykładu klasy encji i jej mapowania. W dalszej części rozdziału omówimy bardziej szczegółowe zagadnienia. Opiszemy problemy wybierania klucza podstawowego, konfigurowania generatorów

kluczy i na koniec przejdziemy do strategii generowania identyfikatorów. Zanim przejdziemy do omówienia pojęcia *tożsamość bazy danych* oraz sposobu zarządzania tożsamością w specyfikacji JPA, wyjaśnimy różnicę pomiędzy tożsamością a równością obiektów w Javie.

4.2.1. Tożsamość a równość w Javie

Deweloperzy Javy rozumieją różnicę między tożsamością a równością obiektów Javy. Tożsamość obiektów (==) to pojęcie zdefiniowane przez maszynę wirtualną Javy. Dwie referencje są identyczne, jeżeli wskazują na tę samą lokalizację w pamięci.

Z kolei równość obiektów, czasami nazywana też *równoważnością*, jest pojęciem zdefiniowanym przez metodę `equals()` klas. Równoważność oznacza, że dwa różne (nieidentyczne) egzemplarze mają tę samą wartość — ten sam stan. Dwa różne egzemplarze klasy `String` są równe, jeżeli reprezentują tę samą sekwencję znaków pomimo tego, że każdy egzemplarz ma własne miejsce w przestrzeni pamięci maszyny wirtualnej (z czytelnikami, którzy są ekspertami w Javie, musimy się zgodzić, że klasa `String` to przypadek specjalny; założmy, że do tego samego stwierdzenia skorzystaliśmy z innej klasy).

Utrwalanie komplikuje ten obraz. W przypadku utrwalania obiektowo-relacyjnego egzemplarz utrwalania jest w pamięci reprezentacją konkretnego wiersza (albo wierszy) tabeli (lub tabel) bazy danych. Wraz z tożsamością i równością w Javie zdefiniujemy tożsamość bazy danych. Mamy teraz trzy metody rozróżniania referencji:

- Obiekty są identyczne, jeżeli zajmują to samo miejsce w pamięci maszyny JVM. Można to sprawdzić za pomocą operatora `a==b`. To pojęcie jest znane jako *tożsamość obiektów*.
- Obiekty są równe, jeżeli mają ten sam stan zgodnie z definicją w metodzie `a.equals(Object b)`. Klasy, które jawnie nie przesłaniają tej metody, dziedziczą implementację zdefiniowaną w klasie `java.lang.Object`, która porównuje tożsamość obiektów za pomocą operatora `==`. To pojęcie jest znane jako *równość obiektów*.
- Obiekty zapisane w relacyjnej bazie danych są identyczne, jeżeli współdzielą tę samą tabelę i wartość klucza głównego. To pojęcie, zmapowane do przestrzeni Javy, jest znane jako *tożsamość bazy danych*.

Przyjrzyjmy się teraz, w jaki sposób tożsamość na poziomie bazy danych łączy się z tożsamością obiektów oraz w jaki sposób można wyrazić tożsamość bazy danych w metadanych mapowania. W ramach przykładu stworzymy mapę encji należącej do modelu dziedziny.

4.2.2. Pierwsza klasa encji i mapowanie

W poprzednim rozdziale nie powiedzieliśmy całej prawdy: sama adnotacja `@Entity` nie wystarczy do zmapowania klasy utrwalania. Potrzebna jest również adnotacja `@Id`, tak jak pokazaliśmy na poniższym listingu.

Listing 4.1. Zmapowana encja Item z właściwością identyfikatora**ŚCIEŻKA:** `/model/src/main/java/org/jpuch/model/simple/Item.java`

```

@Entity
public class Item {

    @Id
    @GeneratedValue(generator = "ID_GENERATOR")
    protected Long id;

    public Long getId() { ← Opcjonalne, ale przydatne
        return id;
    }
}

```

To najprostsza klasa encji oznaczona jako „zdolna do utrwalania” za pomocą adnotacji `@Entity` oraz zawierająca mapowanie `@Id` dla właściwości identyfikatora bazy danych. Domyślnie klasa jest mapowana do tabeli o nazwie `ITEM` w schemacie bazy danych.

Każda klasa encji musi mieć właściwość `@Id`. W ten sposób specyfikacja JPA udostępnia tożsamość bazy danych do aplikacji. Nie pokazujemy właściwości identyfikatora na naszych diagramach. Zakładamy, że każda klasa encji ma identyfikator. W naszych przykładach właściwości identyfikatora zawsze nadajemy nazwę `id`. Jest to dobra praktyka projektowa. Warto stosować tę samą nazwę właściwości identyfikatora dla wszystkich klas encji modelu dziedziny. Jeśli nie wprowadzimy innych specyfikacji, to ta właściwość będzie zmapowana do kolumny klucza głównego o nazwie `ID` tabeli `ITEM` należącej do schematu bazy danych.

Podczas ładowania i zapisywania elementów, aby uzyskać dostęp do wartości właściwości identyfikatora, framework Hibernate skorzysta z pola, a nie z metod gettera lub settera. Ponieważ adnotacja `@Id` dotyczy pola, framework Hibernate domyślnie zezwoli na to, aby każde pole klasy było utrwalaną właściwością. W specyfikacji JPA istnieje następująca reguła: jeżeli zdefiniowano adnotację `@Id` dla pola, to JPA będzie korzystał z pól klasy bezpośrednio i domyślnie uzna wszystkie pola za część utrwalanego stanu. Sposób przesłonięcia tej własności opisaliśmy w dalszej części tego rozdziału. Z naszych doświadczeń wynika, że dostęp do pól jest zwykle najlepszym wyborem, ponieważ pozostawia znacznie więcej swobody podczas projektowania metod dostępowych.

Czy należy zdefiniować (publiczną) metodę gettera dla właściwości identyfikatora? W aplikacjach często wykorzystuje się identyfikatory bazy danych jako wygodne uchwyt określonego egzemplarza — nawet poza warstwą utrwalania. Przykładowo w aplikacjach webowych wyniki wyszukiwania są powszechnie wyświetlane w formie listy podsumowań. Gdy użytkownik wybierze konkretny element, aplikacja może potrzebować pobrania wybranej pozycji. Do tego celu powszechnie wykorzystuje się wyszukiwanie według identyfikatora. Prawdopodobnie już korzystałeś z identyfikatorów w taki sposób — nawet w tych aplikacjach, które bazują na JDBC.

Czy należy zdefiniować metodę settera? Wartości kluczy głównych nigdy się nie zmieniają, dlatego nie należy pozwalać na modyfikowanie wartości właściwości identyfikatora. Framework Hibernate nie aktualizuje kolumny klucza głównego i nie należy udostępniać publicznej metody settera dla identyfikatora encji.

Typ Javy właściwości identyfikatora — w poprzednim przykładzie `java.lang.Long` — zależy od typu kolumny klucza głównego tabeli `ITEM` oraz sposobu tworzenia wartości kluczy. To odsyła nas do adnotacji `@GeneratedValue` i do kluczy głównych w ogóle.

4.2.3. Wybieranie klucza głównego

Identyfikator bazy danych encji jest mapowany na klucz główny jakiejś tabeli, dlatego najpierw spróbujemy omówić tematykę kluczy głównych bez zwracania uwagi na mapowanie. Spróbujemy cofnąć się o krok i zastanowić nad tym, w jaki sposób identyfikujemy encje.

Klucz kandydat jest kolumną (albo zbiorem kolumn), którą można wykorzystać do zidentyfikowania określonego wiersza w tabeli. Aby klucz kandydat mógł zostać kluczem głównym, musi spełnić następujące wymagania:

- Wartością kolumny klucza kandydata nigdy nie jest `null`. Nie można niczego zidentyfikować za pomocą nieznanych danych, dlatego w modelu relacyjnym nie występują wartości `null`. Niektóre produkty SQL pozwalają na definiowanie (złożonych) kluczy głównych z kolumnami, które mogą przyjmować wartości `null`, dlatego należy zachować ostrożność.
- Wartość kolumny (kolumn) klucza kandydata jest unikatowa dla każdego wiersza.
- Wartość kolumny (kolumn) klucza kandydata nigdy się nie zmienia. Jest niemutowalna.

Czy klucze główne muszą być niemutowalne?

Klucz kandydat, zgodnie z definicją modelu relacyjnego, musi być unikatowy i nieredukowalny (żaden podzbiór atrybutów klucza nie może być niepowtarzalny). Poza koniecznością spełnienia tych wymagań wybór klucza kandydata, który ma pełnić funkcję klucza głównego, jest kwestią gustu. Framework Hibernate oczekuje jednak, aby klucz kandydat używany w roli klucza głównego był niezmienny. Hibernate nie wspiera aktualizowania wartości klucza głównego za pośrednictwem API. Przy próbie obejścia tego wymagania możemy napotkać problemy z mechanizmami buforowania i sprawdzania aktualizacji frameworka. Jeżeli schemat bazy danych jest oparty na kluczach głównych dających się aktualizować (i być może używa ograniczeń `ON UPDATE CASCADE`), to trzeba zmodyfikować ten schemat, aby mógł działać z Hibernate.

Jeżeli tabela ma tylko jeden identyfikujący atrybut, to z definicji staje się on kluczem głównym. Jednak w przypadku konkretnej tabeli te funkcje może spełniać kilka kolumn albo kombinacja kilku kolumn. Aby zdecydować o tym, który klucz główny dla tabeli będzie najlepszy, należy dokonać wyboru spośród kluczy kandydatów. Klucze kandydatów niewybrane do roli klucza głównego należy zadeklarować w bazie danych jako unikatowe, jeśli ich wartość jest rzeczywiście unikatowa (ale być może nie jest niezmienna).

W wielu klasycznych modelach danych SQL używane są naturalne klucze główne. *Klucz naturalny* to taki, który ma znaczenie biznesowe: jest atrybutem lub kombinacją atrybutów unikatową ze względu na biznesową semantykę. Przykładami naturalnych kluczy obcych jest numer ubezpieczenia społecznego w Stanach Zjednoczonych lub numer PESEL w Polsce. Rozpoznanie kluczy naturalnych jest bardzo proste: jeżeli

atrybut klucza kandydata ma znaczenie na zewnątrz kontekstu bazy danych, to jest to klucz naturalny bez względu na to, czy jest on generowany automatycznie. Należy pomyśleć o użytkownikach aplikacji: jeżeli pracując z aplikacją i rozmawiając na jej temat, użytkownicy odwołują się do atrybutu klucza, to jest to naturalny klucz (na przykład: „Czy możesz wysłać mi zdjęcie towaru 123-abc”?).

Z doświadczenia wiemy, że naturalne klucze główne zazwyczaj sprawiają problemy. Dobry klucz główny musi być unikatowy, niezmienny i nigdy nie może mieć wartości null. Niewiele atrybutów encji spełnia te wymagania, a te, które to robią, nie mogą być skutecznie indeksowane w bazach danych SQL (choć to jest szczegół implementacji, który nie powinien być czynnikiem decydującym i przemawiającym za jakimś kluczem albo przeciwko niemu). Ponadto trzeba mieć pewność, że definicja klucza kandydata nigdy się nie zmieni przez cały czas życia bazy danych. Zmienianie wartości (albo nawet definicji) klucza głównego i wszystkich kluczy obcych, które się do niego odwołują, jest frustrującym zadaniem. Należy oczekiwać od schematu bazy danych przetrwania dziesięcioleci nawet wtedy, kiedy czas życia aplikacji będzie krótszy.

Co więcej, naturalne klucze kandydatów często można znaleźć, jedynie łącząc kilka kolumn w *złożony* klucz naturalny. Takie złożone klucze, chociaż na pewno właściwe dla niektórych artefaktów schematu (jak tabele łączące w relacji *wiele-do-wielu*), potencjalnie znacznie utrudniają utrzymanie bazy danych, tworzenie zapytań *ad hoc* oraz ewolucję schematu. Tematykę kluczy głównych omówimy w dalszej części tej książki, w punkcie 9.2.1.

Z tych powodów zalecamy posługiwanie się sztucznymi identyfikatorami, nazywanymi również *kluczami zastępczymi*. Klucze zastępcze nie mają żadnego biznesowego znaczenia — mają unikatowe wartości wygenerowane przez bazę danych albo aplikację. Użytkownicy aplikacji w idealnej sytuacji nie wiedzą o istnieniu tych kluczy ani nie odwołują się do ich wartości. Są one wewnętrzną częścią systemu. Wprowadzenie kolumny klucza zastępczego jest także odpowiednie w często spotykanej sytuacji, kiedy nie ma kluczy kandydatów. Inaczej mówiąc: w (prawie) każdej tabeli należącej do schematu powinna występować dedykowana kolumna klucza zastępczego służąca tylko temu celowi.

Istnieje kilka dobrze znanych sposobów generowania wartości kluczy zastępczych. Konfiguruje się je za pomocą wspomnianej wyżej adnotacji `@GeneratedValue`.

4.2.4. Konfigurowanie generatorów kluczy

Do oznaczenia właściwość identyfikatora klasy encji potrzebna jest adnotacja `@Id`. Jeżeli nie użyjemy razem z nią adnotacji `@GeneratedValue`, to dostawca JPA zakłada, że przed zapisaniem egzemplarza klasy zadbamy o utworzenie i przypisanie jego identyfikatora. Nazywamy go identyfikatorem *przypisanym przez aplikację*. Przypisywanie identyfikatora encji ręcznie jest konieczne w przypadku, gdy mamy do czynienia ze starszymi bazami danych i (lub) naturalnymi kluczami głównymi. Więcej na temat tego rodzaju mapowania opowiemy w punkcie 9.2.1, który jest dedykowany temu tematowi.

Zwykle chcemy, aby system wygenerował wartość klucza głównego podczas zapisywania egzemplarza encji, dlatego umieszczamy adnotację `@GeneratedValue` obok adnotacji `@Id`. Specyfikacja JPA standaryzuje kilka strategii generowania wartości za pomocą

wartości typu wyliczeniowego `javax.persistence.GenerationType`, którą wybieramy, przypisując `@GeneratedValue(strategy = ...)`:

- `GenerationType.AUTO` — Hibernate wybiera właściwą strategię na podstawie dialektu SQL skonfigurowanej bazy danych. Jest to równoważne adnotacji `@GeneratedValue()` bez jakichkolwiek ustawień.
- `GenerationType.SEQUENCE` — Hibernate oczekuje w bazie danych sekwencji `HIBERNATE_SEQUENCE` (tworzy ją, jeśli posługujemy się odpowiednimi narzędziami). Sekwencja ta będzie wywoływana osobno przed każdą instrukcją `INSERT`, by wygenerować sekwencyjne wartości liczbowe.
- `GenerationType.IDENTITY` — Hibernate oczekuje w bazie danych specjalnej, auto-inkrementowanej kolumny klucza głównego, która w momencie wykonania instrukcji `INSERT` automatycznie wygeneruje wartość liczbową (Hibernate tworzy taką kolumnę za pomocą języka DDL).
- `GenerationType.TABLE` — Hibernate korzysta z dodatkowej tabeli w schemacie bazy danych, w której są zapisane następne liczbowe wartości kluczy głównych — po jednym wierszu na każdą klasę encji. Wartości z tej tabeli są odczytywane i odpowiednio aktualizowane przed wykonaniem instrukcji `INSERT`. Domyślna nazwa tabeli to `HIBERNATE_SEQUENCES`, natomiast nazwy kolumn to `SEQUENCE_NAME` i `SEQUENCE_NEXT_HI_VALUE` (w wewnętrznej implementacji zastosowano złożony, ale wydajny algorytm generowania największych i najmniejszych wartości; więcej informacji na ten temat później).

Chociaż ustawienie `AUTO` wydaje się wygodne, to zwykle potrzebujemy większej kontroli. Z tego powodu nie powinniśmy na nim polegać, a zamiast niego jawnie skonfigurować strategię generowania klucza głównego. Ponadto większość aplikacji działa z sekwencjami bazy danych, ale czasami zachodzi potrzeba spersonalizowania nazwy i innych ustawień sekwencji. Dlatego zamiast wybierania jednej ze strategii JPA, zalecamy zmapowanie identyfikatora za pomocą adnotacji `@GeneratedValue(generator="ID_GENERATOR")`, tak jak to pokazaliśmy w poprzednim przykładzie.

Jest to niestandardowy generator identyfikatorów. Można teraz skonfigurować ustawienie `ID_GENERATOR` niezależnie od klas encji.

Specyfikacja JPA obejmuje dwie wbudowane adnotacje, z których można skorzystać do skonfigurowania generatorów identyfikowanych przez nazwę: `@javax.persistence.SequenceGenerator` oraz `@javax.persistence.TableGenerator`. Za pomocą tych adnotacji można stworzyć generator z własnymi nazwami sekwencji i tabeli. Jak zwykle bywa w przypadku adnotacji JPA, niestety można ich używać tylko na początku (być może w innym przypadku pustej) klasy, a nie w pliku *package-info.java*.

Własność frameworka Hibernate

Z tego powodu, a także dlatego, że adnotacje JPA nie dają nam dostępu do pełnego zbioru funkcji Hibernate, zalecamy rozwiązanie alternatywne: natywną adnotację `@org.hibernate.annotations.GenericGenerator`. Adnotacja ta obsługuje wszystkie strategie generatora identyfikatorów oraz szczegółów konfiguracji. W przeciwieństwie do raczej

ograniczonych adnotacji JPA, możemy skorzystać z adnotacji Hibernate w pliku *package-info.java*, zazwyczaj umieszczonego w tym samym pakiecie, w którym są umieszczone klasy modelu dziedziny. Rekomendowaną konfigurację zaprezentowaliśmy na poniższym listingu.

Listing 4.2. Generator identyfikatorów frameworka Hibernate skonfigurowany jako metadane na poziomie pakietu

ŚCIEŻKA: */model/src/main/java/org/jpwh/model/package-info.java*

```
@org.hibernate.annotations.GenericGenerator(  
    name = "ID_GENERATOR",  
    strategy = "enhanced-sequence", ← ❶ strategia enhanced-sequence  
    parameters = {  
        @org.hibernate.annotations.Parameter(  
            name = "sequence_name", ← ❷ nazwa sekwencji  
            value = "JPWH_SEQUENCE"  
        ),  
        @org.hibernate.annotations.Parameter(  
            name = "initial_value", ← ❸ wartość początkowa  
            value = "1000"  
        )  
    }  
)  
}
```

Stosowanie tej specyficznej dla Hibernate konfiguracji generatora przynosi następujące korzyści:

- Strategia *enhanced-sequence* ❶ generuje sekwencyjne wartości liczbowe. Jeżeli stosowany dialekt SQL obsługuje sekwencje, to Hibernate skorzysta z sekwencji w bazie danych. Jeśli wykorzystywany system DBMS nie wspiera natywnych sekwencji, to framework Hibernate skorzysta z dodatkowej „tabeli sekwencji”, która symuluje zachowanie sekwencji. W ten sposób zyskujemy rzeczywistą przenośność: przed wykonaniem instrukcji SQL INSERT zawsze można wywołać program generujący. Takiej swobody nie ma na przykład w przypadku stosowania automatycznie inkrementowanych kolumn identyfikatorów, które w momencie wykonania instrukcji INSERT generują wartość zwracaną później do aplikacji.
- Możemy skonfigurować nazwę sekwencji ❷. Hibernate skorzysta z istniejącej sekwencji albo utworzy ją automatycznie podczas generowania schematu SQL. Jeżeli system DBMS nie obsługuje sekwencji, to będzie to nazwa specjalnej „tabeli sekwencji”.
- Możemy rozpocząć od wskazanej wartości początkowej ❸. Dostępność tego ustawienia daje nam przestrzeń do wykorzystywania danych testowych. Przykładowo przy próbie uruchomienia testów integracyjnych Hibernate będzie wprowadzać nowe dane z kodu testów o wartościach identyfikatora większych niż 1000. Dowolne dane testowe, które chcemy zaimportować przed testem, mogą posługiwać się wartościami od 1 do 999, a w testach możemy posługiwać się stabilnymi wartościami identyfikatorów: „załaduj towar o identyfikatorze 123 i uruchom na nim jakieś testy”. Opcja jest stosowana w momencie, gdy Hibernate generuje schemat SQL i sekwencję. Jest to opcja DDL.

Tę samą sekwencję bazy danych możemy współdzielić pomiędzy wszystkimi klasami modelu dziedziny. Nie ma żadnych przeszkód, aby adnotacja `@GeneratedValue(generator = "ID_GENERATOR")` została umieszczona w klasach wszystkich encji. Nie ma znaczenia, czy wartości klucza głównego są ciągle dla jakiejś konkretnej encji, jednak pod warunkiem że są unikatowe w ramach jednej tabeli. W przypadkach gdy ma znaczenie rywalizacja (ponieważ sekwencja musi być wywołana przed każdą instrukcją `INSERT`), można zastosować odmianę omawianej konfiguracji generatora. Omówimy ją później, w podrozdziale 20.1.

W roli typu właściwości identyfikatora w klasie encji używamy klasy `java.lang.Long`, co doskonale pasuje do liczbowego generatora sekwencji bazy danych. Można by również skorzystać z prymitywu `long`. Główna różnica polega na wartości zwracanej przez wywołanie `someItem.getId()` na nowym egzemplarzu, który jeszcze nie został zapisany do bazy danych: `null` albo `0`. Przy sprawdzaniu, czy określony egzemplarz jest nowy, test na wartość `null` jest prawdopodobnie bardziej zrozumiały dla osoby z zewnątrz, która czyta nasz kod. Dla identyfikatorów nie należy stosować innego typu całkowitoliczbowego, takiego jak `int` albo `short`. Chociaż przez pewien czas mogą one działać (być może nawet przez wiele lat), to ze względu na to, że z czasem baza danych będzie się powiększać, możemy być ograniczeni ich zakresem. Jeżeli nowy identyfikator byłby generowany co jedną milisekundę bez żadnych przerw, to rozwiązanie bazujące na typie `int` działałoby przez prawie dwa miesiące, natomiast w przypadku skorzystania z typu `long` wystarczyłoby na około 300 milionów lat.

Strategia `enhanced-sequence` pokazana na listingu 4.2 to zalecana strategia sekwencji dla większości aplikacji. Jest to jednak tylko jedna z wbudowanych strategii dostępnych dla frameworka Hibernate.

Własność frameworka Hibernate

4.2.5. Strategie generatorów identyfikatorów

Poniżej zamieściliśmy listę wszystkich dostępnych strategii identyfikatorów generatorów, ich opcji oraz naszych zaleceń dotyczących ich użycia. Jeżeli nie chcesz teraz czytać całej tej listy, włącz opcję `GenerationType.AUTO` i sprawdź domyślne zachowanie frameworka Hibernate dla wybranego dialektu bazy danych. Istnieje duże prawdopodobieństwo, że strategie `sequence` bądź `identity` będą dobre, ale nie będą to opcje najbardziej wydajne lub przenośne. Jeżeli potrzebujesz spójnego i przenośnego zachowania oraz wartości identyfikatorów dostępnych przed wykonaniem instrukcji `INSERT`, skorzystaj ze strategii `enhanced-sequence` zaprezentowanej w poprzednim punkcie. To przenośna, elastyczna i nowoczesna strategia, która dodatkowo oferuje różne optymalizatory dla dużych zbiorów danych.

Pokażemy również relację pomiędzy każdą ze standardowych strategii JPA a natywnym odpowiednikiem Hibernate. Framework Hibernate rozwijał się organicznie. Z tego powodu mamy teraz dwa zbiory mapowania pomiędzy standardową a natywną strategią — na liście będą one nazywane *Starym* i *Nowym*. Mapowanie można przełączyć za pomocą ustawienia `hibernate.id.new_generator_mappings` w pliku `persistence.xml`.

Generowanie identyfikatorów przed instrukcją INSERT albo po niej: jaka jest różnica?

Usługa ORM próbuje zoptymalizować instrukcje SQL INSERT, na przykład łącząc je w grupy — po kilka — na poziomie JDBC. Dzięki temu wykonanie instrukcji SQL podczas jednostki pracy następuje jak najpóźniej, a nie w chwili wywołania `entityManager.persist(someItem)`. Wywołanie umieszcza jedynie instrukcję wstawiania w kolejce do późniejszego wykonania, a jeśli to możliwe, przypisuje encji wartość identyfikatora. Jednak jeżeli wtedy wywołamy `someItem.getId()`, to możemy uzyskać wartość `null`, jeśli silnik nie zdoła wygenerować identyfikatora przed instrukcją INSERT. Ogólnie rzecz biorąc, preferujemy strategię generowania *pre-insert*. W wyniku zastosowania tych strategii wartości identyfikatorów są generowane niezależnie — przed wykonaniem instrukcji INSERT. Powszechnie wybraną opcją jest współdzielona i dostępna równolegle sekwencja bazy danych. Kolumny autoinkrementowane, domyślne wartości kolumn albo klucze generowane za pomocą wyzwalaczy są dostępne wyłącznie po wykonaniu instrukcji INSERT.

Domyślna wartość to `true`, co odpowiada Nowemu mapowaniu. Oprogramowanie nie starzeje się tak dobrze, jak wino:

- `native` — powoduje automatyczny wybór innych strategii, takich jak `sequence` lub `identity`, w zależności od skonfigurowanego dialektu SQL. Wymaga przeglądania dokumentacji Javadoc (albo nawet kodu źródłowego) dialektu SQL skonfigurowanego w pliku *persistence.xml*. Odpowiednik JPA `GenerationType.AUTO` w Starym mapowaniu.
- `sequence` — wykorzystuje natywną sekwencję bazy danych o nazwie `HIBERNATE_SEQUENCE`. Sekwencja jest wywoływana przed każdą instrukcją INSERT dla nowego wiersza. Możemy spersonalizować nazwę sekwencji oraz wprowadzić dodatkowe ustawienia DDL. Więcej informacji można znaleźć w dokumentacji Javadoc dla klasy `org.hibernate.id.SequenceGenerator`.
- `sequence-identity` — generuje wartości kluczy, wywołując sekwencję bazy danych podczas wstawiania wierszy do bazy danych, na przykład `insert into ITEM(ID) values (HIBERNATE_SEQUENCE.nextval)`. Wartość klucza jest pobierana po wykonaniu instrukcji INSERT. Jest to takie samo zachowanie jak w przypadku strategii `identity`. Pozwala na używanie tych samych parametrów i typów właściwości jak w przypadku strategii `sequence`. Więcej informacji można znaleźć w dokumentacji Javadoc klasy `org.hibernate.id.SequenceIdentityGenerator` oraz jej klasy nadrzędnej.
- `enhanced-sequence` — wykorzystuje natywną sekwencję bazy danych, kiedy to jest możliwe. W przeciwnym razie używana jest dodatkowa tabela bazy danych z pojedynczą kolumną i wierszem, emulująca sekwencję. Domyślna nazwa to `HIBERNATE_SEQUENCE`. „Sekwencja” bazy danych jest wywoływana zawsze przed instrukcją INSERT, dzięki czemu uzyskujemy takie samo zachowanie niezależnie od tego, czy system DBMS obsługuje rzeczywiste sekwencje. Istnieje możliwość skorzystania z optymalizatora `org.hibernate.id.enhanced.Optimizer`, dzięki któremu nie ma konieczności sięgania do bazy danych przed każdą instrukcją INSERT. Ustawienie domyślne to brak optymalizacji i pobieranie nowej wartości dla każdej instrukcji INSERT. Więcej przykładów można znaleźć w rozdziale 20. Opis wszystkich parametrów można znaleźć w dokumentacji Javadoc dla klasy

`org.hibernate.id.enhanced.SequenceStyleGenerator`. Odpowiednik JPA `GenerationType.SEQUENCE` oraz `GenerationType.AUTO` z włączonym Nowym mapowaniem. Jest to najprawdopodobniej najlepsza opcja spośród strategii wbudowanych.

- `seqhilo` — używa natywnej sekwencji bazy danych o nazwie `HIBERNATE_SEQUENCE`; optymalizuje wywołania przed instrukcjami `INSERT`, łącząc wartości `hi` i `lo`. Jeżeli wartość `hi` pobrana z sekwencji to 1, to następne dziewięć instrukcji wstawiania do bazy będzie wykonywanych z wartościami kluczy 11, 12, 13... 19. Później zostaje wywołana kolejna sekwencja w celu uzyskania następnej wartości `hi` (2 albo większej) i procedura jest powtarzana dla wartości 21, 22, 23 itd. Istnieje możliwość skonfigurowania maksymalnej wartości `lo` (9 to ustawienie domyślne) za pomocą parametru `max_lo`. Niestety, ze względu na wadę kodu frameworka Hibernate *nie* można skonfigurować tej strategii za pomocą adnotacji `@GenericGenerator`. Jedynym sposobem skorzystania z tej strategii jest użycie ustawienia JPA `GenerationType.SEQUENCE` i Starego mapowania. Istnieje możliwość skonfigurowania tej strategii za pomocą standardowej adnotacji JPA `@SequenceGenerator` na klasie (która w innym przypadku byłaby pusta). Więcej informacji można uzyskać w dokumentacji Javadoc dla klasy `org.hibernate.id.SequenceHiLoGenerator` i jej klasy nadrzędnej. Zamiast tej strategii warto zastanowić się nad użyciem strategii `enhanced-sequence` z optymalizatorem.
- `hilo` — używa dodatkowej tabeli o nazwie `HIBERNATE_UNIQUE_KEY` oraz tego samego algorytmu, jak w przypadku strategii `seqhilo`. W tabeli występuje pojedyncza kolumna i wiersz z następną wartością sekwencji. Domyślna maksymalna wartość `lo` wynosi 32767, więc warto skonfigurować tę wartość za pomocą parametru `max_lo`. Więcej informacji można znaleźć w dokumentacji Javadoc dla klasy `org.hibernate.id.TableHiLoGenerator`. Nie zalecamy stosowania tej przestarzałej strategii. Zamiast niej lepiej wykorzystać strategię `enhanced-sequence` wraz z optymalizatorem.
- `enhanced-table` — używa dodatkowej tabeli o nazwie `HIBERNATE_SEQUENCES`, domyślnie z jednym wierszem reprezentującym sekwencję i przechowującym następną wartość. Ta wartość jest wybierana i aktualizowana w przypadku, kiedy musi być wygenerowana wartość identyfikatora. Generator można skonfigurować tak, aby zamiast tego wykorzystywał wiele wierszy: po jednym dla każdej sekwencji. Więcej informacji można znaleźć w dokumentacji Javadoc dla klasy `org.hibernate.id.enhanced.TableGenerator`. Strategia ta jest odpowiednikiem JPA `GenerationType.TABLE` z włączonym Nowym mapowaniem. Zastępuje przestarzały, ale podobny generator `org.hibernate.id.MultipleHiLoPerTableGenerator`, który jest odpowiednikiem Starego mapowania dla ustawienia JPA `GenerationType.TABLE`.
- `identity` — obsługuje sekwencję `IDENTITY` i autoinkrementowane kolumny w bazach danych DB2, MySQL, MS SQL Server i Sybase. Wartość identyfikatora dla kolumny klucza głównego będzie generowana dla instrukcji SQL `INSERT` wykonywanej dla wiersza. Nie pozwala na wprowadzanie opcji. Niestety, ze względu na wadę kodu frameworka Hibernate *nie* można skonfigurować tej strategii za pomocą adnotacji `@GenericGenerator`. Jedynym dostępnym sposobem użycia tej strategii jest zastosowanie ustawienia JPA `GenerationType.IDENTITY`

oraz Starego albo Nowego mapowania, dzięki czemu strategia stanie się domyślna dla ustawienia `GenerationType.IDENTITY`.

- `increment` — przy uruchamianiu frameworka Hibernate następuje odczytanie maksymalnych wartości (liczbowo) kolumn klucza głównego dla tabel każdej z encji. Za każdym razem, gdy do tych tabel są wstawiane nowe wiersze, odpowiednia wartość jest zwiększana o jeden. Strategia jest szczególnie skuteczna, gdy w aplikacji Hibernate bez klastrów jest wyłączony dostęp do bazy danych. Nie warto jej jednak używać w żadnym innym scenariuszu.
- `select` — Hibernate nie wygeneruje wartości klucza ani nie uwzględni kolumny klucza głównego w instrukcji `INSERT`. Hibernate oczekuje od systemu DBMS przypisania do kolumny wartości (domyślnej dla schematu albo wygenerowanej za pomocą wyzwalacza) w momencie wstawiania wiersza do bazy danych. Następnie kiedy wiersz zostanie wstawiony do bazy, Hibernate odczytuje kolumnę klucza głównego za pomocą zapytania `SELECT`. Wymagany parametrem jest `key`. Za jego pomocą przekazujemy do instrukcji `SELECT` nazwę właściwości identyfikatora w bazie danych (na przykład `id`). Ta strategia nie jest zbyt wydajna i powinna być stosowana tylko ze starszymi sterownikami JDBC, które nie mogą zwrócić wygenerowanych kluczy bezpośrednio.
- `uuid2` — generuje w warstwie aplikacji unikatowy, 128-bitowy klucz `UUID`. Strategia przydatna w sytuacji, gdy potrzebujemy globalnie unikatowych identyfikatorów dla wielu baz danych (załóżmy, że co noc dane z kilku różnych produkcyjnych baz danych są scalane partiami do archiwum). Identyfikator `UUID` może być kodowany w klasie encji jako `java.lang.String`, `byte[16]` albo `java.util.UUID`. Strategia ta zastępuje starsze strategie `uuid` oraz `uuid.hex`. Można ją skonfigurować za pomocą ustawienia `org.hibernate.id.UUIDGenerationStrategy`. Więcej informacji można znaleźć w dokumentacji Javadoc dla klasy `org.hibernate.id.UUIDGenerator`.
- `guid` — wykorzystuje globalnie unikatowy identyfikator generowany przez bazę danych za pośrednictwem funkcji SQL dostępnej dla baz danych Oracle, Ingres, MS SQL Server i MySQL. Hibernate wywołuje funkcję bazy danych przed wykonaniem instrukcji `INSERT`. Wartość jest mapowana na właściwość identyfikatora `java.lang.String`. Aby uzyskać pełną kontrolę nad generowaniem identyfikatora, można skonfigurować strategię za pomocą adnotacji `@GenericGenerator`, podając w pełni kwalifikowaną nazwę klasy, która implementuje interfejs `org.hibernate.id.IdentityGenerator`.

Podsumowując, nasze zalecenia dotyczące generatora identyfikatora są następujące:

- Ogólnie rzecz biorąc, lepiej stosować strategie generowania *pre-insert*, polegające na niezależnym generowaniu wartości identyfikatorów przed wykonaniem instrukcji `INSERT`.
- Lepiej też wykorzystywać strategię *enhanced-sequence*, która bazuje na użyciu natywnej sekwencji bazy danych, jeśli jest obsługiwana, a jeśli nie, to emuluje sekwencję za pomocą dodatkowej tabeli bazy danych z pojedynczą kolumną i wierszem.

Od tej chwili zakładamy, że właściwości identyfikatorów zostały dodane do klas encji modelu dziedziny, a kiedy zostanie wykonane podstawowe mapowanie każdej encji i jej właściwości identyfikatora, jest wykonywane mapowanie właściwości encji reprezentowanych przez typy wartości. Zagadnieniem mapowania typów wartości zajmijmy się w następnym rozdziale. Warto poczytać o opcjach specjalnych, które pozwalają na uproszczenie i usprawnienie mapowania klas.

4.3. Opcje mapowania encji

Sporządziliśmy mapę klasy utrwalania za pomocą adnotacji `@Entity`, wykorzystując wartości domyślne dla wszystkich innych ustawień, takich jak zmapowana nazwa tabeli SQL. W tym podrozdziale omówimy sposób zarządzania niektórymi opcjami na poziomie klasy:

- wykorzystywanie domyślnego nazewnictwa i konwencji nazewnicznej,
- dynamiczne generowanie kodu SQL,
- stosowanie mutowalności encji.

Niniejszy podrozdział można pominąć, a następnie powrócić do niego później w celu rozwiązania konkretnego problemu.

4.3.1. Zarządzanie nazwami

Spróbujmy najpierw omówić nazewnictwo klas encji i tabel. Jeżeli na poziomie klasy zdolnej do utrwalania wprowadzimy samą tylko adnotację `@Entity`, to domyślna, zmapowana nazwa tabeli będzie taka sama jak nazwa klasy. Zwróć uwagę, że nazwy artefaktów SQL piszemy WIELKIMI LITERAMI. Robimy to jednak tylko w celu ich łatwiejszego odróżnienia od pozostałych zapisów. W rzeczywistości w języku SQL wielkość liter nie ma znaczenia. Tak więc klasa encji Javy `Item` po zmapowaniu będzie miała nazwę `ITEM`. Nazwę tabeli można przesłonić za pomocą adnotacji JPA `@Table`, tak jak pokazaliśmy poniżej.

Listing 4.3. Adnotacja `@Table` przesłania zmapowaną nazwę tabeli

ŚCIEŻKA: `/model/src/main/java/org/jpwh/model/simple/User.java`

```
@Entity
@Table(name = "USERS")
public class User implements Serializable {

    // ...
}
```

Encja `User` zostałaby zmapowana na nazwę tabeli `USER`. W większości systemów DBMS bazujących na SQL jest to słowo zarezerwowane. Nie można stworzyć tabeli o takiej nazwie, dlatego zamiast niej zastosowaliśmy mapowanie do tabeli `USERS`. Adnotacja `@javax.persistence.Table` pozwala także na użycie opcji `catalog` i `schema`, które można wykorzystać w przypadku, gdy baza danych wymaga stosowania takich prefiksów nazw.

Jeżeli naprawdę zachodzi taka konieczność, to można zastosować cudzysłowy, które pozwalają używać zarezerwowanych nazw języka SQL, a nawet posługiwać się nazwami, w których wielkość liter ma znaczenie.

KORZYSTANIE Z CUDZYSŁÓWÓW DLA IDENTYFIKATORÓW SQL

Od czasu do czasu, zwłaszcza w przypadku starszych baz danych, można spotkać identyfikatory z dziwnymi znakami albo spacjami. Czasami chcemy również wymusić rozpoznawanie wielkości liter. Może się też zdarzyć — tak jak w pokazanym wcześniej przykładzie — że automatyczne mapowanie klasy albo właściwości wymagałoby użycia nazwy tabeli lub kolumny, która jest zarezerwowanym słowem kluczowym.

Hibernate 5 rozpoznaje zarezerwowane słowa kluczowe wybranego systemu DBMS na podstawie skonfigurowanego dialektu bazy danych. Potrafi też automatycznie ujmować takie nazwy w cudzysłów podczas generowania SQL. Aby włączyć opcję automatycznego ujmowania w cudzysłów, można skorzystać z opcji `hibernate.auto_quote_keyword = true` w konfiguracji jednostki utrwalania. Jeżeli posługujemy się starszą wersją Hibernate albo informacje o dialekcie okazują się niepełne, należy — w przypadku konfliktu ze słowem kluczowym — zadbać o ręczne wprowadzenie cudzysłówów w definicji mapowania.

Jeśli w definicji mapowania nazwa tabeli lub kolumny zostanie ujęta w lewe apostrofy (*backticks*), to w wygenerowanym SQL framework Hibernate zawsze ujmie ten identyfikator w cudzysłów. Funkcjonalność ta nadal działa w najnowszych wersjach frameworka Hibernate, ale w JPA 2.0 została zestandaryzowana jako *ograniczone identyfikatory* (*delimited identifiers*). Dla nich wykorzystywane są cudzysłowy.

Oto modyfikacja poprzedniego przykładu z wykorzystaniem lewych apostrofów, działająca wyłącznie dla Hibernate:

```
@Table(name = "`USER`")
```

Aby zapewnić zgodność ze specyfikacją JPA, należałoby „unieszkodliwić” cudzysłowy w ciągu znaków:

```
@Table(name = "\"USER\"")
```

Oba te sposoby bezproblemowo działają z frameworkiem Hibernate. Hibernate zna natywny znak „cytatu” dla wybranego dialektu i prawidłowo generuje SQL: `[USER]` dla MS SQL Server, `'USER'` dla MySQL, `"USER"` dla H2 itd.

W przypadku konieczności „cytowania” *wszystkich* identyfikatorów SQL można stworzyć plik *orm.xml* i dodać sekcję `<delimited-identifiers/>` wewnątrz sekcji `<persistence-unit-defaults>`, tak jak to pokazano na listingu 3.8. W takiej sytuacji Hibernate wymusi stosowanie „cytowanych” identyfikatorów wszędzie.

Wszędzie, gdzie to możliwe, należy dążyć do przemianowania nazw tabel i kolumn, dla których wykorzystano nazwy będące zarezerwowanymi słowami kluczowymi. Pisanie zapytań SQL *ad hoc* w konsoli jest trudne w przypadku, gdy wszystkie znaki cytatów i tzw. symbole „ucieczki” (*escape symbols*) trzeba wpisywać ręcznie.

W dalszej części rozdziału pokażemy, w jaki sposób framework Hibernate może pomóc w środowiskach, w których obowiązują ściśle konwencje nazewnnicze dla tabel i kolumn w bazach danych.

Własność frameworka Hibernate

IMPLEMENTACJA KONWENCJI NAZEWNICZEJ

Framework Hibernate dostarcza cechy funkcjonalnej, która pozwala na automatyczne egzekwowanie standardów nazewnniczych. Załóżmy, że wszystkie nazwy tabel w systemie CaveatEmptor powinny mieć postać CE_<nazwa tabeli>. Jednym z rozwiązań jest ręczne wprowadzenie adnotacji @Table na poziomie wszystkich klas encji. To podejście jest czasochłonne i łatwo zapomnieć o tym obowiązku. Zamiast tego rozwiązania można zaimplementować interfejs PhysicalNamingStrategy frameworka Hibernate albo przesłonić istniejącą implementację, tak jak pokazano na poniższym listingu.

Listing 4.4. PhysicalNamingStrategy, przesłanianie domyślnych konwencji nazewnniczych

ŚCIEŻKA: /shared/src/main/java/org/jpwh/shared/CENamingStrategy.java

```
public class CENamingStrategy extends
    org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl {

    @Override
    public Identifier toPhysicalTableName(Identifier name,
                                         JdbcEnvironment context) {
        return new Identifier("CE_" + name.getText(), name.isQuoted());
    }
}
```

Przesłonięcie metody toPhysicalTableName() powoduje poprzedzenie przedrostkiem CE_ wszystkich wygenerowanych nazw tabel w schemacie. Warto zapoznać się z dokumentacją Javadoc interfejsu PhysicalNamingStrategy. Interfejs ten oferuje metody niestandardowego nazewnictwa dla kolumn, sekwencji oraz innych artefaktów.

Implementację strategii nazewnniczej należy włączyć w pliku *persistence.xml*:

```
<persistence-unit>name="CaveatEmptorPU">
  ...
  <properties>
    <property name="hibernate.physical_naming_strategy"
              value="org.jpwh.shared.CENamingStrategy"/>
  </properties>
</persistence-unit>
```

Inny sposób personalizacji nazewnictwa polega na wykorzystaniu interfejsu ImplicitNamingStrategy. O ile fizyczna strategia nazewnicza działa na najniższym poziomie (w momencie generowania nazw artefaktów schematu), o tyle niejawna strategia nazewnicza jest wywoływana wcześniej. Jeżeli mapujemy klasę encji i nie wprowadzamy adnotacji @Table z jawnie określoną nazwą, to system zadaje pytanie o właściwą nazwę do implementacji niejawnej strategii nazewnniczej. Działanie tego mechanizmu jest

zależne od takich czynników jak nazwa encji i nazwa klasy. Framework Hibernate udostępnia kilka strategii implementacji mechanizmu domyślnych nazw zgodnych z systemami tradycyjnymi albo ze specyfikacją JPA. Domyślna strategia to `ImplicitNamingStrategyJpaCompliantImpl`.

Przyjrzyjmy się innemu problemowi — nazewnictwu encji w zapytaniach.

NAZEWNICTWO ENCJI W ZAPYTANIACH

Domyślnie wszystkie nazwy encji są automatycznie importowane do przestrzeni nazw silnika zapytań. Inaczej mówiąc, w ciągach zapytań JPA można posługiwać się krótkimi nazwami klas bez prefiksu w postaci pakietu, co jest wygodne:

```
List result = em.createQuery("select i from Item i")
    .getResultList();
```

Takie rozwiązanie działa jedynie wtedy, kiedy w jednostce utrwalania występuje tylko jedna klasa `Item`. Jeżeli dodamy inną klasę `Item` w innym pakiecie, to aby było możliwe korzystanie z formy krótkich nazw w zapytaniach JPA, powinniśmy zmienić nazwę jednej z tabel `Item`:

```
package my.other.model;

@javax.persistence.Entity(name = "AuctionItem")
public class Item {

    // ...
}
```

Zapytanie z wykorzystaniem formy krótkiej dla klasy `Item` z pakietu `my.other.model` ma teraz postać `select i from AuctionItem i`. W ten sposób rozwiązaliśmy konflikt nazwy z inną klasą `Item` w innym pakiecie. Oczywiście zawsze możemy użyć w pełni kwalifikowanych, długich nazw z prefiksem pakietu.

Na tym zakończymy naszą podróż po opcjach nazewnictwa frameworka Hibernate. W dalszej części tego rozdziału opowiemy, jak Hibernate generuje kod SQL zawierający te nazwy.

Własność frameworka Hibernate

4.3.2. Dynamiczne generowanie SQL

Domyślnie Hibernate tworzy instrukcje SQL dla każdej klasy utrwalania w momencie tworzenia jednostki utrwalania, przy starcie aplikacji. Są to proste operacje **CRUD** (*create, read, update, delete*) do czytania pojedynczego wiersza, usuwania wiersza itp. Mniej kosztowne niż generowanie ciągów SQL za każdym razem, gdy występuje potrzeba uruchomienia takiego prostego zapytania, jest zapisanie tych zapytań w pamięci „z góry”. W dodatku buforowanie preparowanych instrukcji SQL na poziomie JDBC jest znacznie skuteczniejsze, jeżeli jest mniej instrukcji.

W jaki sposób Hibernate może utworzyć „z góry” instrukcję `UPDATE`? W końcu w tym momencie kolumny do uaktualnienia nie są znane. Otóż wygenerowana instrukcja SQL

uaktualnia wszystkie kolumny, a jeżeli wartość określonej kolumny nie została zmodyfikowana, to instrukcja ustawia ją na starą wartość.

W niektórych sytuacjach (na przykład dla odziedziczonej tabeli bazy danych zawierającej setki kolumn), kiedy instrukcje SQL są bardzo rozbudowane nawet dla najprostszych operacji (powiedzmy, że tylko jedna kolumna wymaga aktualizacji), należy wyłączyć opcję generowania SQL przy starcie i przełączyć się na dynamicznie generowane instrukcje w fazie działania aplikacji. Bardzo duża liczba encji może także wpływać na czas rozruchu, ponieważ Hibernate musi z góry wygenerować wszystkie instrukcje SQL dla operacji CRUD. Konsumpcja pamięci dla takiego bufora instrukcji w przypadku konieczności zbuforowania dziesiątek instrukcji dla dziesiątek encji także będzie wysoka. Może to być problem w przypadku środowisk wirtualnych z ograniczeniami albo w przypadku urządzeń o małej mocy obliczeniowej.

Aby wyłączyć generowanie instrukcji SQL INSERT i UPDATE przy starcie, potrzebne są natywne adnotacje Hibernate:

```
@Entity
@org.hibernate.annotations.DynamicInsert
@org.hibernate.annotations.DynamicUpdate
public class Item {
    // ...
}
```

Włączając dynamiczne instrukcje wstawiania i aktualizacji, zlecamy frameworkowi Hibernate generowanie ciągów instrukcji SQL wtedy, gdy są one potrzebne, a nie z góry. Instrukcja UPDATE będzie zawierać tylko kolumny z aktualizowanymi wartościami, a instrukcja INSERT tylko te kolumny, które nie przyjmują wartości `null`.

Zagadnienie generowania i personalizowania SQL omówimy w rozdziale 17. Czasami, gdy określona encja jest niezmienna, można całkowicie uniknąć generowania instrukcji UPDATE.

Własność frameworka Hibernate

4.3.3. Encje niezmiennie

Egzemplarze określonej klasy mogą być niezmiennie. Przykładowo w systemie CaveatEmptor niezmienny jest egzemplarz klasy `Bid` odpowiadający ofercie złożonej na towar. Z tego powodu Hibernate nigdy nie musi uruchamiać instrukcji UPDATE na tabeli `BID`. Hibernate może także wprowadzić kilka innych optymalizacji — na przykład unikanie sprawdzania modyfikacji dla mapowania klasy niezmiennnej, tak jak pokazano w następnym przykładzie. W tym przykładzie klasa `Bid` jest niezmienna, a jej egzemplarze nigdy nie są modyfikowane:

```
@Entity
@org.hibernate.annotations.Immutable
public class Bid {
    // ...
}
```


Obiekt POJO jest niezmienny, jeżeli nie udostępnia żadnych metod setterów dla żadnych właściwości klasy — wszystkie wartości są ustawiane w konstruktorze. Framework Hibernate podczas ładowania i zapisywania egzemplarzy powinien sięgać bezpośrednio do pól. Mówiliśmy o tym we wcześniejszej części tego rozdziału: jeżeli adnotację `@Id` wprowadzono dla pola, to framework Hibernate będzie bezpośrednio sięgał do pól. Metody getterów i setterów można zaprojektować według potrzeb. Należy także pamiętać, że nie wszystkie frameworki działają z obiektami POJO bez setterów. Przykładowo JSF w celu wypełnienia egzemplarza nie sięga do pól bezpośrednio.

Jeśli nie możemy utworzyć widoku w schemacie bazy danych, możemy zmapować niezmienną klasę encji na zapytanie SQL `SELECT`.

Własność frameworka Hibernate

4.3.4. Mapowanie encji na podzapytanie

Czasami administrator bazy danych nie pozwala zmieniać schematu. Nawet dodanie nowej perspektywy może nie być możliwe. Załóżmy, że chcemy utworzyć widok zawierający identyfikator aukcji towaru `Item` oraz liczbę ofert złożonych na ten towar.

Za pomocą adnotacji Hibernate możemy utworzyć widok poziomu aplikacji — klasę encji tylko do odczytu zmapowaną na instrukcję SQL `SELECT`:

ŚCIEŻKA: `/model/src/main/java/org/jpwh/model/advanced/ItemBidSummary.java`

```
@Entity
@org.hibernate.annotations.Immutable
@org.hibernate.annotations.Subselect(
    value = "select i.ID as ITEMID, i.ITEM_NAME as NAME, " +
            "count(b.ID) as NUMBEROFBIDS " +
            "from ITEM i left outer join BID b on i.ID = b.ITEM_ID " +
            "group by i.ID, i.ITEM_NAME"
)

@org.hibernate.annotations.Synchronize({"Item", "Bid"})
public class ItemBidSummary {

    @Id
    protected Long itemId;

    protected String name;

    protected long numberOfBids;

    public ItemBidSummary() {
    }

    // Metody getterów...

    // ...
}
```

wielkość liter w nazwach tabel ma znaczenie (błąd Hibernate HHH-8430)

Kiedy jest ładowany egzemplarz klasy `ItemBidSummary`, Hibernate uruchamia niestandardową instrukcję SQL `SELECT` jako podzapytanie:

ŚCIEŻKA: `/examples/src/test/java/org/jpwh/test/advanced/MappedSubselect.java`

```
ItemBidSummary itemBidSummary = em.find(ItemBidSummary.class, ITEM_ID);
// select * from (
// select i.ID as ITEMID, i.ITEM_NAME as NAME, ...
// ) where ITEMID = ?
```

Wszystkie nazwy tabel wymienione w instrukcji `SELECT` należy wymienić w adnotacji `@org.hibernate.annotations.Synchronize` (w czasie, kiedy powstawała ta książka, framework Hibernate zawierał błąd HHH-8430¹, polegający na tym, że w nazwach tabel była rozpoznawana wielkość liter).

Hibernate będzie wówczas wiedział, że powinien zatwierdzić modyfikacje egzemplarzy `Item` i `Bid`, zanim uruchomi zapytanie do tabeli `ItemBidSummary`:

ŚCIEŻKA: `/examples/src/test/java/org/jpwh/test/advanced/MappedSubselect.java`

```
Item item = em.find(Item.class, ITEM_ID);
item.setName("Nowa nazwa");

// ItemBidSummary itemBidSummary = em.find(ItemBidSummary.class, ITEM_ID);
Query query = em.createQuery(
    "select ibs from ItemBidSummary ibs where ibs.itemId = :id"
);
ItemBidSummary itemBidSummary =
    (ItemBidSummary)query.setParameter("id", ITEM_ID).getSingleResult();
```

Brak zatwierdzania zmian przed pobraniem rekordu według identyfikatora

Automatyczne zatwierdzanie zmian przed zapytaniami, w przypadku gdy zaktualizowano zsynchronizowane tabele

Zwróć uwagę, że Hibernate nie zatwierdza zmian automatycznie przed wykonaniem operacji `find()`, a dopiero przed uruchomieniem zapytania, jeśli zachodzi taka potrzeba. Hibernate wykryje, że zmodyfikowany egzemplarz klasy `Item` będzie miał wpływ na wynik zapytania ze względu na to, że tabela `ITEM` jest zsynchronizowana z klasą `Item` ↪ `BidSummary`. Z tego powodu, aby nie dopuścić do tego, by zapytanie zwróciło nieaktualne dane, potrzebne jest zatwierdzenie zmian i aktualizacja wiersza tabeli `ITEM`.

4.4. Podsumowanie

- Encje to gruboziarniste klasy w systemie. Ich egzemplarze mają niezależny cykl życia i własną tożsamość. Może się do nich odwoływać wiele innych egzemplarzy klas.
- Z kolei typy wartości zależą od konkretnej klasy encji. Egzemplarz typu wartości jest powiązany z egzemplarzem encji będącym jego właścicielem. Może się do niego odwoływać tylko jeden egzemplarz encji. Typ wartości nie ma własnej tożsamości.

¹ Patrz <https://hibernate.atlassian.net/browse/HHH-8430>.

- Omówiliśmy tematykę tożsamości w Javie, równości obiektów oraz tożsamości na poziomie bazy danych, a także określiliśmy, jakie pola są dobrymi kluczami głównymi. Poznaliśmy generatory wartości kluczy głównych dostarczane przez framework Hibernate, pokazaliśmy, jak z nich korzystać oraz w jaki sposób rozszerzać system identyfikatorów.
- Omówiliśmy przydatne opcje mapowania klas, strategie nazewnicze oraz dynamiczne generowanie SQL.

Skorowidz

A

ACID, 300

adapter typu, 140

adnotacja

 @Access, 130

 @AttributeOverrides,
 133

 @Convert, 143, 146

 @DiscriminatorFormula,
 163

 @ElementCollection,
 184, 509

 @Embeddable, 129, 143

 @Embedded, 131

 @Entity, 79, 98

 @GenericGenerator, 219

 @Id, 101

 @Inject, 373

 @javax.persistence.
 OrderBy, 190

 @JoinColumn, 221

 @JoinTable, 231

 @Lob, 138

 @ManyToOne, 202

 @MapsId, 246

 @NotNull, 129

 @OneToMany, 204

 @org.hibernate.

 annotations.

 CollectionId, 196

 @org.hibernate.

 annotations.OrderBy,

 190

 @PersistenceContext,
 536

 @PostPersist, 373

 @PostLoad, 374

 @PostPersist, 374

 @PostRemove, 374

 @PostUpdate, 374

 @PrePersist, 374

 @PreRemove, 374

 @PreUpdate, 374

 @SecondaryTable, 266

 @Temporal, 125

 @Transactional, 576

agregacja, 443, 475

aktualizowanie

 danych, 607

 egzemplarzy encji, 597

 partiami, 605

aktywacja rejestrowania

 audytu, 382

algorytm pobierania partiami,
 350

aliasy, 425

analiza

 aplikacji bezstanowej, 546

 aplikacji stateful, 554

 dziedziny biznesowej, 64

API, 47, 70

 AuditReader, 385

 Criteria, 473

 CriteriaQuery, 400

 dynamicznego

 metamodelu, 87

 EntityGraph, 332

 EntityManager, 486

 Java Persistence, 563

 JDBC, 484

 Query, 398

 Session, 296

aplikacja

 CaveatEmptor, 62, 457

 JAX-RS, 585

 JSF, 572

 klient-serwer, 529

 webowa, 557

aplikacje

 analiza, 546, 554

 edycja przedmiotu

 aukcji, 540, 549

 kontekst utrwalania

 w konwersacji, 579

 kończenie długotrwałych

 konwersacji, 583

 przepływ pracy

 konwersacji, 577

 rozpoczynanie

 długotrwałych

 konwersacji, 582

 sekwencja wywołań, 543,
 549

 składanie oferty, 543

 usługi o zasięgu

 konwersacji, 576

 usługi o zasięgu żądania,
 572

apt, 90

architektura

 warstwowa, 62

 współdzielonej pamięci

 podręcznej, 609

asocjacje, 43

 dwukierunkowe, 202, 232

 encji, 200

 jeden-do-jednego, 214,
 220

 jeden-do-wielu, 202, 224,
 230

 w klasach osadzanych,
 230

 z tabeląłączenia, 228

 jednokierunkowe, 220,

 232

 POJO, 73

 polimorficzne, 173

asocjacje
 pomiędzy encjami, 179,
 241
 potrójne, 232
 trójelementowe
 z komponentami, 238
 wiele-do-jednego, 201,
 202
 wiele-do-wielu, 232
 dwukierunkowe, 232
 jednokierunkowe, 232
 z pośrednią encją, 234
 z ukrytym
 polimorfizmem, 177
 audyt, 382
 tworzenie śladu, 384
 włączanie rejestrowania,
 383
 autozatwierdzanie, 326

B

bazy danych
 schemat, 127, 180
 Bean Validation, 80
 bezpieczeństwo wątków, 536
 BLOB, 139
 blokady
 offline, 322
 pesymistyczne, 320
 rozszerzanie zakresu, 323
 blokowanie danych, 320
 bootstrap API, 57
 brudny odczyt, 307
 budowa serwera
 bezstanowego, 540
 budowanie
 aplikacji JSF, 572
 aplikacji webowych, 557
 serwera stateful, 548
 buforowanie, 612
 danych, 608
 encji, 616
 kolekcji, 616
 w kontekście utrwalania,
 285

C

CDI, Contexts and
 Dependency Injection, 557
 CLOB, 139
 CRUD, 483
 cykl życia
 egzemplarzy encji, 269,
 276
 utrwalania, 272

D

dane historyczne, 386
 DAO, Data Access Object,
 527, 532
 DBMS, database management
 systems, 33
 DDL, data definition
 language, 34
 definicje typów, 152
 definiowanie
 dynamicznych filtrów, 390
 programowe zapytań, 415
 zapytań
 w metadanych XML,
 413
 z adnotacjami, 414
 deklarowanie profili
 pobierania, 356
 deskryptor XML JPA, 83
 diagram UML, 37
 długotrwałe konwersacje, 583
 DML, Data Manipulation
 Language, 34, 484
 dodawanie ograniczeń, 252
 domyślne wartości
 właściwości, 124
 dostęp
 do API, 55
 do danych, 325
 do historycznych danych,
 386
 do metadanych, 87
 do statystyk, 622
 do właściwości, 120
 współbieżny, 306
 dostosowywanie
 kodu SQL, 483
 mapowania wyników, 490
 operacji CRUD, 505

drobnoziarniste modele
 dziedziny, 94
 drzewo kategorii, 502
 DTO, data transfer objects,
 439
 dynamiczne
 filtry danych, 389
 generowanie SQL, 111
 pobieranie zachłanne, 353
 pobieranie ze
 złączeniami, 453
 tworzenie egzemplarzy,
 439
 dyskryminator, 167
 dziedziczenie, 40, 155
 klas osadzanych, 169

E

edycja zdjęć, 579
 egzemplarz
 typu encji, 95
 typu wartości, 95
 egzemplarze encji
 odłączanie, 294
 scalanie, 295
 EJB, 70
 eksternalizacja
 natywnych zapytań, 502
 zapytań, 412
 encje, 93
 niezależne, 215
 niezmiennie, 112

F

filtr
 włączanie, 391
 zastosowanie, 390
 filtrowanie
 danych, 363, 389
 dostępu do kolekcji, 392
 kolekcji, 470
 filtry dynamiczne, 389
 framework
 CDI, 557
 JSF, 557
 funkcje
 agregacji
 avg(), 443

count(), 443
 max(), 443
 min(), 443
 sum(), 443
 obsługi zapytań
 frameworka Hibernate,
 435
 JPA, 434

G

generator
 identyfikatorów, 104
 kluczy, 101
 obcych, 218
 głównych, 217
 generowanie
 opcji klucza obcego, 210
 schematu, 249
 SQL, 111
 generyczne interfejsy DAO,
 532
 globalne metadane, 79
 graf encji, 357
 grupowanie, 445

H

HATEOAS, 586
 hermetyzacja, 191
 Hibernate
 EntityManager, 50
 Envers, 50, 382
 łatwość utrzymania, 46
 niezależność od
 producenta, 46
 OGM, 50
 ORM, 49
 produktywność, 46
 Search, 50
 Validator, 50
 hierarchia klas, 160
 historyczne egzemplarze encji,
 387

I

identyfikator, 457
 bazy danych, 127
 iloczyn kartezjański, 346, 456

implementacja
 asocjacji POJO, 73
 generycznego interfejsu,
 533
 interceptorów, 376
 interfejsu UserType, 148
 konwencji nazewnictwa,
 110
 metody równości, 291,
 292

 modelu dziedziny, 66
 obiektów DAO encji, 536
 POJO, 69
 stronicowania, 567, 568

indeksy, 257

informacje

 o atrybucie encji, 88

 o typie encji, 87

inicjalizacja obiektów proxy, 334

inkrementacja wersji, 319

instrukcja

 SELECT, 344, 345, 352

 UPDATE, 597

instrukcje

 CUDA, 510

 masowe, 596

 SQL, 52

instrumentacja kodu

 bajtowego, 138

integracja JPA z CDI, 558

integralność relacji, 76

inteligentne wrappery, 333

interceptor, 377

interfejs

 DAO, 532

 EntityManager, 276

 java.io.Serializable, 585

 kolekcji, 182

 org.hibernate.Interceptor
 , 377

 StatelessSession, 606

 TypedQuery, 398

 UserType, 149

interfejsy

 API, 302

 zapytań

 frameworka Hibernate,
 402

 JPA, 398

iterowanie po wynikach, 411

izolacja transakcji, 307

 NONSTRICT_READ_
 WRITE, 614

 READ_ONLY, 614

 READ_WRITE, 613

 TRANSACTIONAL, 613

J

jawne pesymistyczne blokady,
 320

JAXB, Java Architecture for
 XML Binding, 587

JDBC, Java Database
 Connectivity, 35

jednostka

 pracy, 277

 utrwalania, 51, 52

język

 definicji danych, DDL, 34

 DML, 484

 HQL, 399

 JPQL, 47, 332, 399, 596

 manipulowania danymi,
 DML, 34

 język SQL, 34

JMX, Java Management
 Extension, 622

JPA, 45

JPQL, Java Persistence Query
 Language, 47, 56, 332, 596

JSF, JavaServer Faces, 70, 557

K

kanoniczna jednostka pracy,
 276

kaskadowe

 odświeżanie, 368

 przejścia stanu, 364

 replikacje, 370

 usuwanie, 206

 zmiany stanu, 204

klasa, 39

 Address, 215

 BankAccount, 165

 BillingDetails, 156, 158,
 163

 CreditCard, 157, 159, 165

- klasa
 - Dimensions, 170
 - Item, 171
 - javax.persistence.Query, 398
 - javax.persistence.TypedQuery, 398
 - Metamodel, 425
 - PersistenceUtil, 332
 - ResultTransformer, 469
 - User, 69
 - Weight, 170
 - klasy
 - osadzane, 96, 169, 230
 - utrwalania, 53, 66
 - klauszula
 - FROM, 424
 - SELECT, 424
 - klient, 540
 - JVM, 550
 - klucz
 - główny, 100
 - naturalny, 100, 258
 - złożony, 259
 - obcy
 - odwołujący się do pól, 265
 - w kluczach głównych, 261, 264
 - klucze
 - mapy, 198
 - odziedziczone, 258
 - kolejkowanie modyfikacji, 328
 - kolejność
 - iteracji, 190
 - par klucz-wartość, 191
 - kolekcja
 - Map, 241
 - Set komponentów, 194
 - kolekcje
 - bag, 180, 224
 - ciągów znaków, 185
 - jeden-do-wielu, 224
 - komponentów, 196
 - bez typów generycznych, 182
 - komponentów, 191
 - posortowane, 188
 - uporządkowane, 188
 - w komponencie osadzonym, 199
 - kolumna
 - @JoinColumn, 230
 - złączenia klucza obcego, 220
 - komponent
 - bean, 551
 - stateful, 551
 - komponenty
 - jako klucze mapy, 198
 - osadzane, 126, 128, 130
 - komunikaty o błędach
 - walidacji, 252
 - konfigurowanie, *Patrz także*
 - ustawianie
 - generatorów kluczy, 101
 - jednostki utrwalania, 51
 - przez wyjątek, 118
 - współdzielonej pamięci podręcznej, 614
 - kontekst utrwalania, 274
 - w konwersacji, 579
 - kontrolowanie
 - wygenerowanego schematu, 250
 - konwencje nazewnictwa, 110
 - konwersacje, 540, 550
 - długotrwałe, 582
 - konwersja
 - prostych wartości
 - właściwości, 142
 - właściwości
 - komponentów, 144
 - konwerter, 134
 - JPA, 140
 - cursor, 521
 - kwantyfikacja, 461
- L**
- leniwe ładowanie
 - asocjacji, 336
 - trwałych kolekcji, 337
 - z przechwytywaniem, 339
 - licznik aktualizacji, 516
 - lista, 180, 186
 - dwukierunkowa
 - z mappedBy, 227
 - logika trójwartościowa, 428
 - logowanie instrukcji SQL, 52
- Ł**
- ładowanie
 - drzewa, 503
 - komunikatów, 54
 - leniwe, 332
 - zachłanne, 332
 - łączenie obiektu
 - EntityManager z transakcjami, 560
- M**
- mapa wartości komponentów, 197
 - mapowanie, 91
 - abstrakcyjnej klasy
 - bazowej, 156
 - aliasów na właściwości
 - JavaBean, 468
 - asocjacji encji, 200, 365
 - dla dziedziczenia, 155
 - duplikatów pól, 493
 - encji, 108
 - na podzapytanie, 113
 - z tożsamością, 97
 - klas hierarchii, 161
 - do tabeli, 164
 - klas utrwalania, 93
 - klasy
 - BankAccount, 165
 - BillingDetails, 156, 158, 163
 - CreditCard, 159, 165
 - Dimensions, 170
 - Item, 171
 - Weight, 170
 - kolekcji, 179
 - bag identyfikatorów, 185
 - komponentów
 - osadzanych, 126
 - list, 186
 - dwukierunkowych, 226
 - jednokierunkowych, 226

mapy, 187
 mieszane, 499
 na klasy encji, 488
 naturalnych kluczy
 głównych, 258
 obiektowo-relacyjne,
 ORM, 29
 osadzalnej klasy bazowej,
 169
 podklasy, 157
 pól na właściwości
 komponentów, 495
 encji, 490
 prostych właściwości, 118
 relacji wiele-do-wielu, 235
 typów wartości, 117
 typów wyliczeniowych,
 126
 unii, 158
 wielkości skalarnych, 499
 właściwości, 266
 będących kolekcjami,
 180
 wyników, 490
 zapytania SQL, 486
 wyniku na konstruktor,
 498
 za pomocą konwerterów,
 134
 zaawansowane asocjacje,
 213
 zagnieżdżonych
 komponentów
 osadzanych, 132
 zbioru, 184
 złożonych kluczy
 głównych, 259
 mapy, 187
 typów wartości, 180
 masowe instrukcje
 JPQL, 597
 w SQL, 601
 mechanizm
 ładujący, 506
 z procedurą, 523
 MVCC, 307
 specyfikowania
 metadanych
 mapowania, 47

mechanizmy konfiguracji
 Hibernate, 57
 metadane
 adnotacji globalnych, 79
 bazujące na adnotacjach,
 78
 modelu dziedziny, 77
 odwzorowania jednostki
 utrwalania, 83
 XML, 413
 XML z JPA, 83
 metamodel statyczny, 88
 metoda
 between(), 429
 concat(), 441
 convertToDatabaseColumn(), 145
 delete(), 606
 EntityManager#persist(), 603
 equals(), 192, 291
 getReference(), 333
 getResultList(), 408
 gt(), 429
 hashCode(), 192, 291
 insert(), 606
 isNull(), 429
 isValid(), 545
 placeBid(), 576
 sessionWithOptions(), 379
 String#compareTo(), 189
 update(), 606
 metody przesunąć, 564
 mieszane strategie
 dziedziczenia, 167
 model dziedziny aplikacji, 65
 moduł hibernate-
 entitymanager, 51
 modyfikowanie trwałych
 danych, 279
 MVCC, 307

N

naruszenie ograniczeń, 82
 narzędzie
 apt, 90
 Arquillian, 539

bytecode enhancer, 342
 nasłuchiwanie zdarzeń, 372
 natywne mechanizmy
 konfiguracji, 57
 nazewnictwo encji
 w zapytaniach, 111
 niedopasowanie
 paradygmatów, 37
 niejawna obsługa polimorfizmu,
 156
 niepowtarzalność wyników, 440
 niezmienna klasa typu, 141
 nowe własności JPA 2, 118
 numer wersji, 316, 385

O

obiekt
 Connection, 485
 CriteriaBuilder, 467
 dostępu do danych, 532
 EntityManager, 279, 536,
 551, 558
 POJO, 128
 obiekty, 98
 bazy danych, 247
 o dużych rozmiarach, 138
 proxy, 589
 proxy encji, 333
 transferu danych, DTO,
 439
 obserwatory zdarzeń, event
 listeners, 364, 372
 obsługa
 kluczy odziedziczonych,
 258
 polimorfizmu, 156
 współbieżności, 312
 wyjątków, 303
 zdarzeń, 380
 odczyt
 danych, 326
 widmo, 308
 odłączanie egzemplarzy encji,
 294
 odpowytywanie strona po stronie,
 571
 odświeżanie, 368
 danych, 284

ograniczanie, 474
 walidacji, 81
 ograniczenia
 bazy danych, 255
 konwerterów JPA, 146
 SQL, 250
 ograniczenie poziomu tabeli, 254
 opcja
 on delete cascade, 210
 ResultCheckStyle, 525
 opcje
 kaskadowe, 365
 mapowania encji, 108
 przechodniego
 utrwalania, 371
 serializacji, 139
 złączenia, 449
 operacje
 CRUD, 32, 111, 483, 522
 CUD, 507, 524
 DML, 507
 na kolekcjach, 509
 złączeń, 446
 operator
 CASE, 423
 COALESCE, 423
 NULLIF, 423
 TREAT, 423
 operatory porównań, 428
 optymalizacja ślepego strzału, 349
 ORM, object/relational
 mapping, 29, 31, 45
 osadzana klasa, 132

P

pamięć podręczna
 architektura, 609
 drugiego poziomu, 604, 610
 konfigurowanie, 614
 testowanie, 619
 ustawianie trybów, 622
 wybór strategii
 współbieżności, 613
 wyników zapytania, 624
 zarządzanie, 623
 parametry
 nazwane, 404
 pozycyjne, 406
 wejściowe, 518
 wyjściowe, 518
 pary klucz-wartość, 187
 personalizacja
 dostępu do właściwości, 120
 operacji na kolekcjach, 509
 pesymistyczne blokowanie
 danych, 320
 pierwsza
 aplikacja, 50
 klasa encji, 98
 pierwszeństwo operatorów
 JPQL, 432
 plik
 hbm.xml, 54
 import.sql, 249
 package-info.java, 79, 103, 152
 persistence.xml, 51, 104, 248
 pliki
 mapowania XML, 85
 XML, 83
 pobieranie
 asocjacji, 368
 danych partiami, 349
 danych z
 wyprzedzeniem, 349
 dynamiczne, 453
 kolekcji z
 wyprzedzeniem, 351
 metadanych, 83
 pojedynczego wyniku, 408
 referencji, 281
 trwałych danych, 279
 zachłanne, 352, 496, 511
 podklasy, 164
 odpowiedź
 do zapytań, 416
 nazwanych zapytań, 419
 ustawianie
 komentarza SQL, 419
 limitu czasu, 417
 rozmiaru pobierania, 418
 trybu synchronizacji, 417
 trybu tylko do odczytu, 418
 podtypy, 40
 podzapytania, 351, 459, 478
 polecenie apt, 90
 polimorficzne
 asocjacje wiele-do-jednego, 173
 kolekcje, 176
 polimorfizm, 156
 porównywanie
 identyfikatorów, 457
 poruszanie się po danych, 44
 porządkowanie, 473
 wartości null, 436
 poziomy izolacji
 Czytaj niezatwierdzone, 309
 Czytaj zatwierdzone, 309
 Izolacja szeregowalna, 309
 Powtarzalny odczyt, 309
 problem
 iloczynu kartezjańskiego, 346
 n+1, 345
 podtypów, 40
 poruszania się po danych, 44
 tożsamości, 41
 z asocjacjami, 43
 ziarnistości, 38
 procedura utrwalania
 partiami, 603
 procedury
 dla operacji CUD, 524
 składowane, 513, 522
 profile pobierania, 355, 361
 programowe definiowanie
 nazwanych zapytań, 415
 projektowanie aplikacji, 529
 proste typy właściwości, 96
 proxy, 333, 589
 przechodnie
 odłączanie, 366
 utrwalanie, 371
 przechwytywanie zdarzeń, 372

przełączanie
 danych, 282
 strategii, 169
 przepływ pracy konwersacji,
 577
 przesłanianie
 osadzonych atrybutów,
 131
 strategii dostępu, 120
 przesłonięcie, 194
 przesunięcie, 563
 przetwarzanie
 adnotacji, 90
 masowe, 596
 wsadowe, 596, 602
 przewijalne zbiory wyników,
 409
 przewijanie kursorów
 zwracanych, 522
 przypisywanie aliasów, 425
 punkty rozszerzeń, 147

R

rdzeń zapytania, 425
 referencja, 95, 281
 null, 130
 rejestrowanie audytu, 383
 relacja, 75
 jeden-do-jednego, 201
 jeden-do-wielu z
 kluczem, 241
 wiele-do-jednego, 201
 wiele-do-wielu, 201
 relacje trójczone
 klucz-wartość, 243
 relacyjne bazy danych, 33
 replikacje, 370
 replikowanie danych, 284
 reprezentowanie nazwy pliku,
 198
 ręczne zarządzanie wersjami,
 317
 RMI, Remote Method
 Invocation, 585
 rozgraniczanie transakcji, 306
 rozszerzanie
 frameworka, 147
 zakresu blokowania, 323
 rozszerzenia producentów, 78

równość, 98
 egzemplarzy
 komponentu, 192
 rzutowanie, 437, 475
 encji, 437
 wartości skalarnych, 437
 z zapytaniami SQL, 487

S

scalanie, 366
 egzemplarzy encji, 295
 schemat bazy danych, 127, 180
 schematy
 odziedziczone, 245
 złożone, 245
 sekwencja wywołań, 543, 549
 selekcja, 424, 473
 serializacja, 139
 danych, 585
 obiektów proxy, 589
 serwer bezstanowy, 540
 silnik JPA, 52
 skalowanie Hibernate, 595
 skrypty SQL, 247
 słowo kluczowe final, 485
 sortowanie, 188, 436, 562
 kolekcji, 471
 sprawdzanie aktualizacji, 56
 SQL, 483
 stan egzemplarza encji
 odłączony, 274, 288
 przejściowy, 273, 282
 trwałe, 273
 usunięty, 274
 standard
 ANSI, 134
 EJB, 70
 JAX-RS, 587
 stosowanie reguł Bean
 Validation, 80
 strategia
 JOINED, 165
 SINGLE_TABLE, 161
 TABLE_PER_CLASS,
 158
 strategie
 dziedziczenia, 167
 generatorów
 identyfikatorów, 104

mapowania
 dziedziczenia, 172
 pobierania, 344
 współbieżności, 613
 stronicowanie, 562
 dużych zbiorów, 406
 na bazie przesunięć, 563,
 567
 na bazie przeszukiwania,
 563, 568
 w warstwie utrwalania,
 565
 synchronizacja
 frameworka, 329
 kontekstu utrwalania, 287

T

tabele
 pomocnicze, 266
 rejestru audytu, 383
 złączenia, 221, 228
 tablice, 183
 technologia RMI, 585
 testowanie, 82, 545
 instrukcji SQL, 486
 warstwy utrwalania, 538
 współdzielonej pamięci
 podręcznej, 619
 tożsamość, 41, 97
 odłączonych
 egzemplarzy, 289
 transakcje, 299, 300
 bazodanowe i
 systemowe, 301
 programowe z JTA, 301
 transakcyjne
 przetwarzanie danych,
 269
 systemy plików, 184
 transformacje
 wartości kolumn, 123
 wyników zapytań, 466
 tryb JDBC, 484
 tryby blokowania danych, 323
 tworzenie
 egzemplarzy encji, 599
 indeksów, 257
 interceptorów cyklu
 życia, 372

tworzenie
konwerterów JPA, 140
obiektu EntityManager, 558
śladu audytu, 384
warstwy utrwalania, 530
zapytań, 398

typ
ArrayList, 190
LinkedHashMap, 191
LinkedHashSet, 189

typy
binarne, 138
danych użytkownika, 251
dat i godzin, 136
proste i numeryczne, 134
SQL, 134
użytkownika, 147
wartości, 93, 96
wbudowane, 134
wyliczeniowe, 126
znakowe, 135

U

UDT, user-defined data types, 39

ulepszanie schematu bazy
danych, 246

unie, 158

uruchamianie zapytań, 408

usługa JAX-RS, 585

usługi
o zasięgu konwersacji, 576
o zasięgu żądania, 572

ustawianie
komentarza SQL, 419
limitu czasu, 417
rozmiaru pobierania, 418
trybów pamięci
podręcznej, 622
trybu synchronizacji, 417
trybu tylko do odczytu, 418

usuwanie
egzemplarzy encji, 597
egzemplarzy
osieroconych, 208
referencji, 209

utrwalanie, 32
danych, 278
obiektów, 47
przechodnie, 205
przezroczyste, 68
stanu, 205
zautomatyzowane, 68

używanie właściwości
wyprowadzonych, 122

W

warstwa utrwalania, 530, 538

wartości właściwości, 124
wygenerowane przez
bazę, 124

wartość
null, 130, 436
nullable, 177

wersjonowanie, 311-316, 382
encji, 311

wiązanie parametrów
nazwanych, 404

własne
instrukcje CUD, 510
mechanizmy ładujące, 505
operacje DML, 507

własności specyfikacji JPA 2, 246

właściwości, 118
o typach prostych, 119
opisujące czas, 125
tylko do odczytu, 122

właściwość
billingAddress, 131
java.util.Collection, 183
java.util.List, 183
java.util.Map, 183
java.util.Set, 182
java.util.SortedMap, 183
noop, 121

włączanie wersjonowania, 315

wsparcie dla typów UDT, 39

współbieżność, 299, 307, 312, 612

współdzielenie klucza
głównego, 214

współdzielone referencje, 97

wstawianie egzemplarzy encji
partiami, 603

wstrzykiwanie
obiektu EntityManager, 560
SQL, 403

wybór
adaptera typu, 140
interfejsu kolekcji, 182
klucza głównego, 100
poziomu izolacji, 310
strategii, 172
pobierania, 344
współbieżności, 613

wygenerowane wartości
właściwości, 124

wyjątek, 303
ConstraintViolation
↳Exception, 304
DataException, 304
HibernateException, 304
JDBCConnection
↳Exception, 304
JDBCException, 304
LockAcquisition
↳Exception, 304
LockTimeoutException, 305
NonUniqueResult
↳Exception, 305
NoResultException, 305
QueryTimeoutException, 305
SQLGrammarException, 304

wykrywanie stanu encji, 279

wymuszanie integralności
relacji, 76

wyniki zapytania, 401

wyrażenia
porównań, 428
z kolekcjami, 432

wyszukiwanie granic strony, 571

wyścig, 548

wywołania zwrotne
cyklu życia, 364
JPA, 372

wywoływanie

funkcji, 433
 funkcji w projekcjach,
 441
 procedur składowanych,
 513
 zapytania przez nazwę,
 412
 wzorzec obiektu dostępu do
 danych, 532

Z

zachłanne
 ładowanie asocjacji, 342,
 368
 ładowanie kolekcji, 342,
 496
 pobieranie, 353, 511
 zagnieżdżanie
 nieskorelowane, 460
 skorelowane, 460
 zakleszczenia, 324
 zależności cyklu życia, 97
 zapisywanie komunikatów, 54
 zapytania, 395, 397, 403
 eksternalizacja, 412
 nazywanie, 412
 niepolimorficzne, 426
 ograniczenia, 427
 opcje zaawansowane, 465
 podpowiedzi, 416

polimorficzne, 426
 pomocnicze, subselects,
 34
 przez przykład, 479
 transformacje wyników,
 466
 tworzenie, 398
 uruchamianie, 408
 w metadanych XML, 413
 z adnotacjami, 414
 z rzutowanymi
 elementami, 466
 zarządzanie
 danymi, 271
 nazwami, 108
 niezależnymi
 egzemplarzami encji,
 205
 relacjami, 75
 wersjami, 317
 współbieżnością, 312
 współbieżnym dostępem,
 306
 współdzieloną pamięcią
 podręczną, 623
 zasięg
 przepływu JSF, 577
 tożsamości obiektu, 289
 zbiór, 180, 184
 osadzanych
 komponentów, 194

zdarzenie, 372
 ładowania, 381
 ziarnistość, 38
 złączenia, 164, 221, 446, 477
 jawne, 451
 niejawne asocjacyjne,
 449
 teta, 456
 w JPA, 449
 w SQL, 447
 złączenie
 INNER JOIN, 354
 LEFT OUTER JOIN,
 355
 zmiana stanu encji, 204
 znacznik czasu, 314
 zwracanie
 kursora, 521
 listy list, 466
 listy map, 467
 wielu zbiorów wyników,
 516
 zbioru wyników, 515

Ż

żądanie
 GET, 585
 PUT, 585

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**

Utrwalenie danych, tak aby zostały one zapisane i przechowane przez system informatyczny, jest jedną z podstawowych funkcji aplikacji. Prawie zawsze wymaga się trwałych danych. W przypadku Javy zazwyczaj utrwalenie danych odbywa się poprzez ich zapisanie w relacyjnej bazie danych z wykorzystaniem języka SQL. Relacyjne bazy danych stanowią niezwykle elastyczne i potężne narzędzie do zarządzania utwalonymi danymi, jednak aby wykorzystać wszystkie zalety tego rozwiązania, trzeba zapewnić optymalne komunikowanie się aplikacji z bazą danych.

Niniejsza książka stanowi wyczerpujące źródło aktualnej wiedzy o frameworku Hibernate, najpopularniejszym narzędziu do utrwalania danych dla Javy, które zapewnia automatyczne i przejrzyste mapowanie obiektowo-relacyjne. Wyczerpująco opisano też standard Java Persistence 2.1 (JSR 338). Programowanie aplikacji w Hibernate wyjaśniono tu na licznych przykładach. Pokazano, jak korzystać z mapowania, zapytań, strategii pobierania, transakcji, konwersacji, buforowania i wielu innych funkcji. Nie zabrakło opisu najlepszych praktyk w projektowaniu baz danych oraz wskazówek dotyczących optymalizacji. Wszystkie przykłady zostały uaktualnione dla najnowszych wersji frameworka Hibernate i środowiska Java EE.

Najważniejsze zagadnienia omówione w książce:

- Mapowanie obiektowo-relacyjne i jego znaczenie
- Projekt aplikacji bazodanowej typu klient-serwer
- Korzystanie z frameworka Hibernate
- Specyfikacja Java Persistence
- Transakcyjne przetwarzanie danych, w tym zagadnienie współbieżności
- Tworzenie i uruchamianie zapytań oraz przetwarzanie otrzymanych wyników

Christian Bauer jest szkoleniowcem i konsultantem. Bierze udział w rozwijaniu frameworka Hibernate. Autor kilku książek dotyczących programowania w Javie. **Gavin King** jest współzałożycielem projektu Hibernate oraz członkiem grupy ekspertów pracujących nad standardem Java Persistence (JSR 220). Przewodził również pracom nad standaryzacją CDI (JSR 299). **Gary Gregory** jest współautorem książek *JUnit in Action* oraz *Spring Batch in Action*. Jest także członkiem grup zarządzania projektami firmy Apache Software Foundation: Commons, HttpComponents, Logging Services i Xalan.

Hibernate i Java Persistence
— najlepszy sposób na nowoczesną aplikację bazodanową!

Helion

księgarnia internetowa



<http://helion.pl>

zamówienia telefoniczne



0 801 339900



0 601 339900

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

Sprawdź najnowsze promocje:
• <http://helion.pl/promocje>
Książki najchętniej czytane:
• <http://helion.pl/bestsellery>
Zamów informacje o nowościach:
• <http://helion.pl/novosci>

Informatyka w najlepszym wydaniu

sięgnij po WIĘCEJ



KOD KORZYŚCI

ISBN 978-83-283-2782-5



9 788328 327825

cena: 99,00 zł