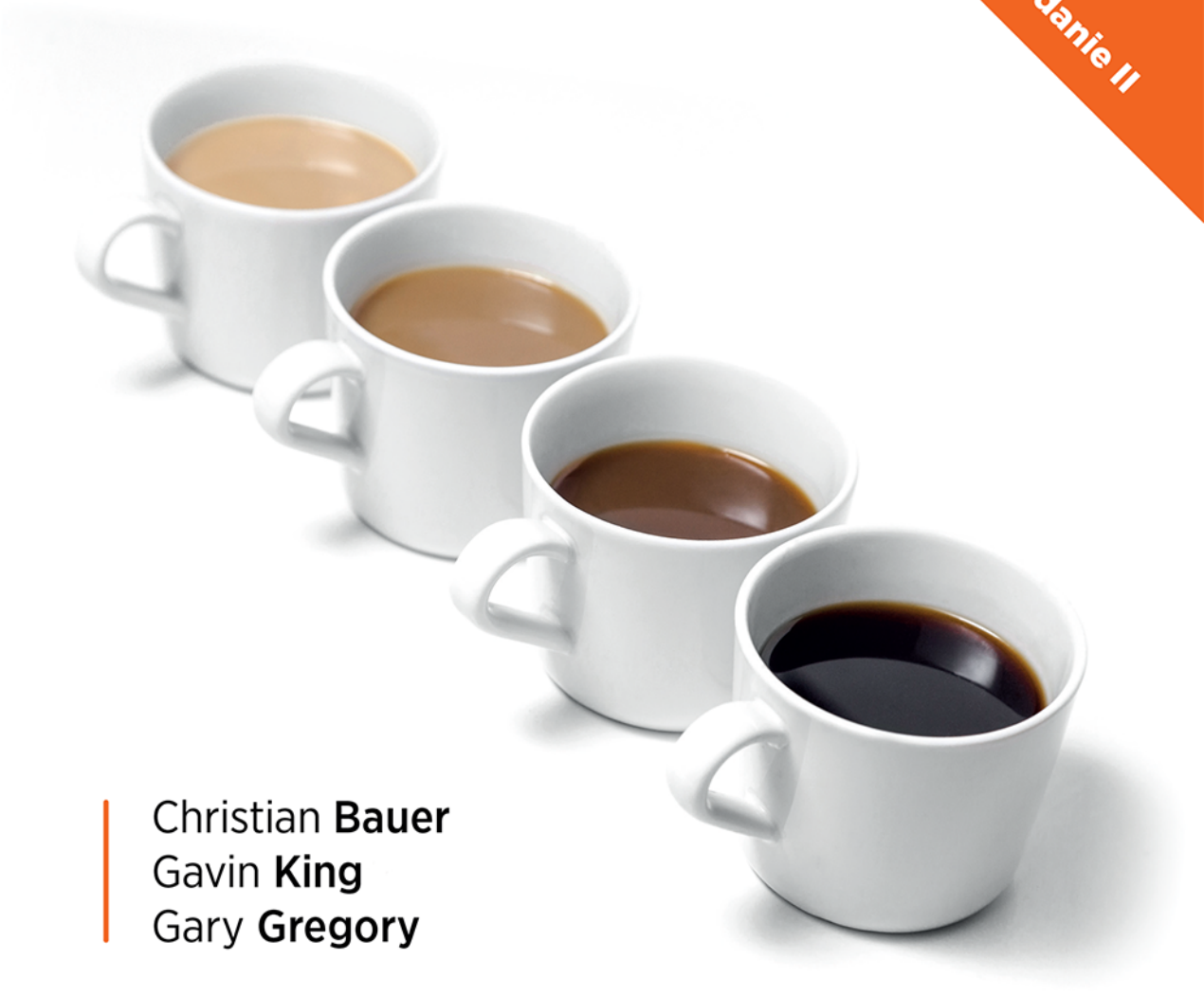
Wydanie II



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

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.

Spis treści

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 wysokopoziomową 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.

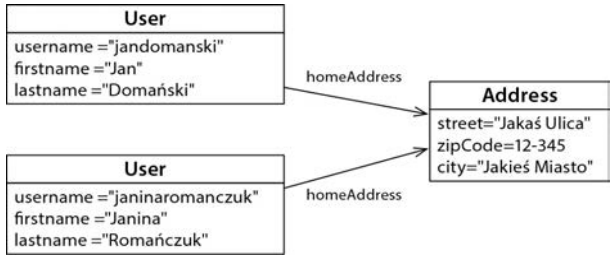
4.1.2. Definiowanie pojęć aplikacji

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.

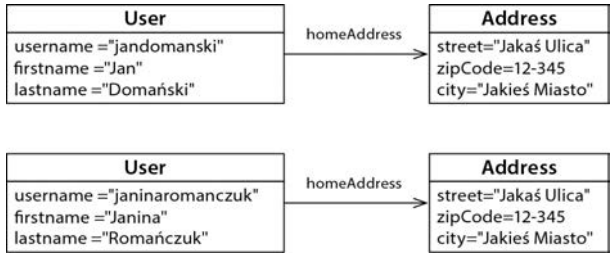
Klasa `User` ma właściwość `homeAddress`. Jest to asocjacja z klasą `Address`. Czy oba egzemplarze klasy `User` mają w fazie działania aplikacji referencje do tego samego egzemplarza klasy `Address`, czy też każdy egzemplarz klasy `User` ma referencję do własnego egzemplarza klasy `Address`? Czy ma znaczenie 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:

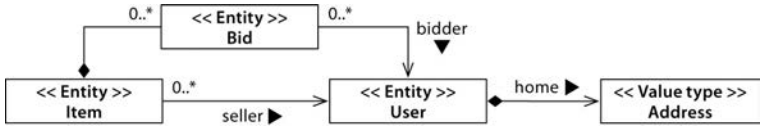
- Możemy pobrać egzemplarz *typu encji*, używając jego tożsamości utrwalania: na przykład egzemplarza `User`, `Item` albo `Category`. Referencja do egzemplarza encji (wskaźnika w JVM) jest utrwalana jako referencja w bazie danych (wartość z ograniczeniem klucza obcego). Egzemplarz encji ma własny cykl 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 osadzanymi* (*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; załóżmy, 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/jpwh/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 uchwyty 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. Spróbujmy 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 niereduko-walny (żaden podzbiór atrybutów klucza nie może być niepowtarzalny). Poza koniecz-nością 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 zdjęcia 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 (chociaż 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, autoinkrementowanej 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",   <-- (1) strategia enhanced-sequence
parameters = {
    @org.hibernate.annotations.Parameter(
        name = "sequence_name",   <-- (2) nazwa sekwencji
        value = "JPWH_SEQUENCE"
    ),
    @org.hibernate.annotations.Parameter(
        name = "initial_value",   <-- (3) 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 ciągłe 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 strategie generowania *pre-insert*. W wyniku zastosowania tych strategii wartości identyfikatorów są generowane niezależnie — przed wykonaniem instrukcji `INSERT`. Powszechnie wybieraną 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 Generation ↪Type.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 hii 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 @Sequence ↪Generator na klasie (która w innym przypadku byłaby pusta). Więcej informacji można uzyskać w dokumentacji Javadoc dla klasy org.hibernate.id.SequenceHiLo ↪Generator 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 Generation ↪Type.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 wyłączny 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`. Wymaganym 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 zajmiemy 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 nazewniczej,
- 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 CUDZYSŁOWÓW 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 cudzysłowów 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 ścisłe konwencje nazewnicze dla tabel i kolumn w bazach danych.

Własność frameworka Hibernate

IMPLEMENTACJA KONWENCJI NAZEWNICZEJ

Framework Hibernate dostarcza cechy funkcjonalnej, która pozwala na automatyczne egzekwowanie standardów nazewniczych. 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 nazewniczych

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 nazewniczej 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 nazewniczej. 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 `Implicit ➥NamingStrategyJpaCompliantImpl`.

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 niezmienne

Egzemplarze określonej klasy mogą być niezmienne. 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 niezmiennej, 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 zapamiętać, 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-8430[1], 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 synchronizowana 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

B

C

D

E

F

L

Ł

M

N

O

P

R

S

T

U

W

Z

Ż

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 utrwalonymi 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 przezroczyste 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!

ISBN 978-83-283-2782-5

Informatyka w najlepszym wydaniu

cena: 99,00 zł