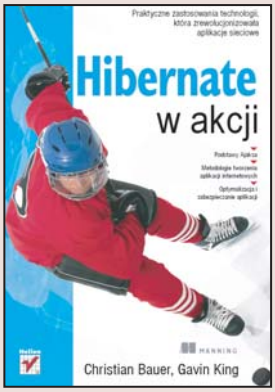
Hibernate w akcji



Autorzy: Christian Bauer, Gavin King
Tłumaczenie: Rafał Jońca
ISBN: 83-246-0527-4
Tytuł oryginału: Hibernate in Action
Format: B5, stron: 410

Szukasz rozwiązania problemów związanych z korzystaniem z relacyjnych baz danych w połączeniu z programowaniem obiektowym? Chcesz poprawić wydajność i jakość aplikacji bazodanowych? A może rozwiązania, które stosowałeś dotychczas, okazują się niewystarczające przy dużych projektach? Sięgnij po Hibernate, czyli rewolucyjne narzędzie stanowiące warstwę pośredniczącą pomiędzy aplikacją i bazą danych, umożliwiające utrwalanie i odczyt obiektów Javy w bazie. Hibernate eliminuje konieczność ręcznego tworzenia kodu odwzorowującego obiekty na model relacyjny i odwrotnie, a także znacząco poprawia wydajność i stabilność aplikacji. Nie bez znaczenia również jest fakt, iż Hibernate dostępne jest na licencji open-source.

Książka „Hibernate w akcji", napisana przez twórców tego narzędzia. To wyczerpujący podręcznik dla programistów, którzy planują zastosować je w swoich projektach. Czytając ją, dowiesz się, na czym polega odwzorowanie obiektowo-relacyjne i w jaki sposób implementuje je Hibernate. Poznasz zasady tworzenia i stosowania obiektów trwałych, zarządzania transakcjami i buforowania danych. Znajdziesz także informacje o optymalizowaniu wydajności aplikacji stosujących Hibernate oraz procesie projektowania takich aplikacji.

W książce poruszono m.in.:

- Odwzorowania obiektowo-relacyjne
- Konfiguracja i uruchomienie Hibernate
- Odwzorowywanie danych w klasach
- Stosowanie obiektów trwałych
- Transakcje i buforowanie
- Wydajne pobieranie obiektów z bazy
- Projektowanie aplikacji wykorzystujących Hibernate
- Narzędzia wspomagające działanie Hibernate

Poznaj Hibernate i przekonaj się, jak dzięki niemu usprawnisz swoją pracę

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

Przedmowa 9

Wstęp 11

Podziękowania 13

O książce i autorach 15

O Hibernate 3 i EJB 3 19

1. Trwałość dzięki odwzorowaniu obiektowo-relacyjnemu 21

1.1. Czym jest trwałość? 23

1.1.1. Relacyjne bazy danych 23

1.1.2. Język SQL 24

1.1.3. Korzystanie z SQL w Javie 24

1.1.4. Trwałość w aplikacjach obiektowych 25

1.2. Niedopasowanie paradygmatów 27

1.2.1. Problem szczegółowości 28

1.2.2. Problem podtypów 29

1.2.3. Problem identyczności 30

1.2.4. Problemy dotyczące asocjacji 32

1.2.5. Problem nawigacji po grafie obiektów 33

1.2.6. Koszt niedopasowania 34

1.3. Warstwy trwałości i alternatywy 35

1.3.1. Architektura warstwowa 35

1.3.2. Ręczne tworzenie warstwy trwałości za pomocą SQL i JDBC 37

1.3.3. Wykorzystanie serializacji 38

1.3.4. A może ziarenka encyjne EJB? 39

1.3.5. Obiektowe systemy bazodanowe ... 40
1.3.6. Inne rozwiązania ... 41
1.4. Odwzorowanie obiektowo-relacyjne ... 41
1.4.1. Czym jest ORM? ... 42
1.4.2. Ogólne problemy ORM ... 44
1.4.3. Dlaczego ORM? ... 45
1.5. Podsumowanie ... 47

2. Wprowadzenie i integracja Hibernate 49

2.1. „Witaj świecie" w stylu Hibernate ... 50
2.2. Podstawy architektury Hibernate ... 55
2.2.1. Interfejsy podstawowe ... 56
2.2.2. Interfejsy wywołań zwrotnych ... 58
2.2.3. Typy ... 58
2.2.4. Interfejsy rozszerzeń ... 59
2.3. Konfiguracja podstawowa ... 59
2.3.1. Tworzenie obiektu SessionFactory ... 60
2.3.2. Konfiguracja w środowisku niezarządzanym ... 62
2.3.3. Konfiguracja w środowisku zarządzanym ... 66
2.4. Zaawansowane ustawienia konfiguracyjne ... 68
2.4.1. Konfiguracja bazująca na pliku XML ... 69
2.4.2. Obiekt SessionFactory dowiązany do JNDI ... 70
2.4.3. Dzienniki ... 71
2.4.4. Java Management Extensions ... 73
2.5. Podsumowanie ... 75

3. Odwzorowanie klas trwałości danych 77

3.1. Aplikacja CaveatEmptor ... 78
3.1.1. Analiza dziedziny biznesowej ... 79
3.1.2. Model dziedzinowy CaveatEmptor ... 79
3.2. Implementacja modelu dziedzinowego ... 82
3.2.1. Kwestia przesiąkania zadań ... 82
3.2.2. Trwałość automatyczna i przezroczysta ... 83
3.2.3. Tworzenie klas POJO ... 84
3.2.4. Implementacja asocjacji POJO ... 86
3.2.5. Dodanie logiki do metod dostępowych ... 90
3.3. Definicja metadanych odwzorowujących ... 92
3.3.1. Metadane w pliku XML ... 92
3.3.2. Podstawowe odwzorowania właściwości i klas ... 95
3.3.3. Programowanie zorientowane na atrybuty ... 101
3.3.4. Modyfikacja metadanych w trakcie działania aplikacji ... 102
3.4. Identyczność obiektów ... 104
3.4.1. Identyczność a równość ... 104
3.4.2. Tożsamość bazodanowa w Hibernate ... 104
3.4.3. Wybór kluczy głównych ... 106
3.5. Szczegółowe modele obiektów ... 108
3.5.1. Encje i typy wartości ... 109
3.5.2. Stosowanie komponentów ... 109
3.6. Odwzorowanie dziedziczenia klas ... 113
3.6.1. Tabela na klasę konkretną ... 113
3.6.2. Tabela na każdą hierarchię klas ... 115

3.6.3. Tabela na każdą podklasę ... 117
3.6.4. Wybór strategii ... 120
3.7. Asocjacje ... 121
3.7.1. Asocjacje zarządzane? ... 121
3.7.2. Krotność ... 122
3.7.3. Najprostsza możliwa asocjacja ... 122
3.7.4. Tworzenie asocjacji dwukierunkowej ... 123
3.7.5. Związek rodzic-potomek ... 126
3.8. Podsumowanie ... 127

4. Stosowanie obiektów trwałych 129

4.1. Cykl życia obiektu trwałego ... 130
4.1.1. Obiekty ulotne ... 131
4.1.2. Obiekty trwałe ... 132
4.1.3. Obiekt odłączony ... 133
4.1.4. Zasięg identyczności obiektów ... 134
4.1.5. Poza zasięgiem identyczności ... 135
4.1.6. Implementacja equals() i hashCode() ... 136
4.2. Zarządca trwałości ... 140
4.2.1. Czynienie obiektu trwałym ... 140
4.2.2. Aktualizacja stanu trwałego obiektu odłączonego ... 141
4.2.3. Pobranie obiektu trwałego ... 142
4.2.4. Aktualizacja obiektu trwałego ... 143
4.2.5. Zmiana obiektu trwałego na ulotny ... 143
4.2.6. Zmiana obiektu odłączonego na ulotny ... 144
4.3. Trwałość przechodnia w Hibernate ... 144
4.3.1. Przechodniość przez osiągalność ... 145
4.3.2. Trwałość kaskadowa w Hibernate ... 146
4.3.3. Zarządzanie kategoriami przedmiotów ... 147
4.3.4. Rozróżnienie obiektów ulotnych i odłączonych ... 151
4.4. Pobieranie obiektów ... 152
4.4.1. Pobieranie obiektów na podstawie identyfikatora ... 153
4.4.2. Wprowadzenie do HQL ... 154
4.4.3. Zapytania przez określenie kryteriów ... 155
4.4.4. Zapytanie przez przykład ... 155
4.4.5. Strategie sprowadzania danych ... 156
4.4.6. Wybór strategii sprowadzania w odwzorowaniach ... 158
4.4.7. Optymalizacja pobierania obiektów ... 163
4.5. Podsumowanie ... 164

5. Transakcje, współbieżność i buforowanie 167

5.1. Transakcje bazodanowe ... 169
5.1.1. Transakcje JDBC i JTA ... 170
5.1.2. Interfejs Transaction ... 171
5.1.3. Opróżnianie sesji ... 173
5.1.4. Poziomy izolacji ... 174
5.1.5. Wybór poziomu izolacji ... 176
5.1.6. Ustawianie poziomu izolacji ... 177
5.1.7. Blokada pesymistyczna ... 177
5.2. Transakcje aplikacyjne ... 180
5.2.1. Wersjonowanie zarządzane ... 181

5.2.2. Szczegółowość sesji ... 184
5.2.3. Inne sposoby implementacji blokady optymistycznej ... 185
5.3. Buforowanie — teoria i praktyka ... 186
5.3.1. Strategie i zasięgi buforowania ... 187
5.3.2. Architektura buforów Hibernate ... 190
5.3.3. Buforowanie w praktyce ... 195
5.4. Podsumowanie ... 204

6. Zaawansowane zagadnienia odwzorowań 205

6.1. System typów Hibernate ... 206
6.1.1. Wbudowane typy odwzorowań ... 207
6.1.2. Zastosowania typów odwzorowań ... 210
6.2. Odwzorowywanie kolekcji typów wartości ... 220
6.2.1. Zbiory, pojemniki, listy i odwzorowania ... 220
6.3. Odwzorowanie asocjacji encyjnych ... 228
6.3.1. Asocjacja jeden-do-jednego ... 229
6.3.2. Asocjacje wiele-do-wielu ... 232
6.4. Odwzorowanie asocjacji polimorficznych ... 241
6.4.1. Polimorficzna asocjacja wiele-do-jednego ... 241
6.4.2. Kolekcje polimorficzne ... 243
6.4.3. Asocjacje polimorficzne i jedna tabela na klasę konkretną ... 244
6.5. Podsumowanie ... 246

7. Wydajne pobieranie obiektów 247

7.1. Wykonywanie zapytań ... 249
7.1.1. Interfejsy zapytań ... 249
7.1.2. Dowiązywanie parametrów ... 251
7.1.3. Zapytania nazwane ... 254
7.2. Proste zapytania dotyczące obiektów ... 255
7.2.1. Najprostsze zapytanie ... 256
7.2.2. Zastosowanie aliasów ... 256
7.2.3. Zapytania polimorficzne ... 257
7.2.4. Ograniczenia ... 257
7.2.5. Operatory porównań ... 258
7.2.6. Dopasowywanie tekstów ... 260
7.2.7. Operatory logiczne ... 261
7.2.8. Kolejność wyników zapytań ... 262
7.3. Złączanie asocjacji ... 262
7.3.1. Złączenia w Hibernate ... 264
7.3.2. Pobieranie asocjacji ... 265
7.3.3. Aliasy i złączenia ... 266
7.3.4. Złączenia niejawne ... 270
7.3.5. Złączenia w stylu theta ... 271
7.3.6. Porównywanie identyfikatorów ... 272
7.4. Tworzenie zapytań raportujących ... 273
7.4.1. Projekcja ... 274
7.4.2. Agregacja ... 276
7.4.3. Grupowanie ... 277
7.4.4. Ograniczanie grup klauzulą having ... 278
7.4.5. Poprawa wydajności zapytań raportujących ... 279

7.5. Techniki tworzenia zaawansowanych zapytań ... 279
7.5.1. Zapytania dynamiczne ... 280
7.5.2. Filtry kolekcji ... 282
7.5.3. Podzapytania ... 284
7.5.4. Zapytania SQL ... 286
7.6. Optymalizacja pobierania obiektów ... 288
7.6.1. Rozwiązanie problemu n+1 pobrań danych ... 288
7.6.2. Zapytania iterate() ... 291
7.6.3. Buforowanie zapytań ... 292
7.7. Podsumowanie ... 294

8. Tworzenie aplikacji stosujących Hibernate 295

8.1. Projektowanie aplikacji warstwowych ... 296
8.1.1. Użycie Hibernate w systemie serwletowym ... 297
8.1.2. Stosowanie Hibernate w kontenerze EJB ... 311
8.2. Implementacja transakcji aplikacyjnych ... 319
8.2.2. Trudny sposób ... 321
8.2.3. Odłączone obiekty trwałe ... 322
8.2.4. Długa sesja ... 323
8.2.5. Wybór odpowiedniej implementacji transakcji aplikacyjnych ... 327
8.3. Obsługa specjalnych rodzajów danych ... 328
8.3.1. Starsze schematy baz danych i klucze złożone ... 328
8.3.2. Dziennik audytowy ... 337
8.4. Podsumowanie ... 343

9. Narzędzia Hibernate 345

9.1. Procesy tworzenia aplikacji ... 346
9.1.1. Podejście z góry na dół ... 347
9.1.2. Podejście z dołu do góry ... 347
9.1.3. Podejście od środka ... 347
9.1.4. Spotkanie w środku ... 347
9.1.5. Ścieżka przejścia ... 348
9.2. Automatyczne generowanie schematu bazy danych ... 348
9.2.1. Przygotowanie metadanych odwzorowania ... 349
9.2.2. Utworzenie schematu ... 351
9.2.3. Aktualizacja schematu bazy danych ... 353
9.3. Generowanie kodu klas POJO ... 354
9.3.1. Wprowadzenie metaatrybutów ... 355
9.3.2. Metody odnajdujące ... 357
9.3.3. Konfiguracja hbm2java ... 358
9.3.4. Uruchamianie narzędzia hbm2java ... 359
9.4. Istniejące schematy i Middlegen ... 360
9.4.1. Uruchomienie Middlegen ... 360
9.4.2. Ograniczanie tabel i związków ... 362
9.4.3. Dostosowanie generowania metadanych ... 363
9.4.4. Generowanie metadanych hbm2java i XDoclet ... 365
9.5. XDoclet ... 366
9.5.1. Ustawianie atrybutów typu wartości ... 367
9.5.2. Odwzorowanie asocjacji encyjnych ... 368
9.5.3. Uruchomienie XDoclet ... 369
9.6. Podsumowanie ... 371

A Podstawy języka SQL 373

B Strategie implementacji systemów ORM 377

B.1. Właściwości czy pola? 378
B.2. Strategie sprawdzania zabrudzenia 379
B.2.1. Dziedziczenie po wygenerowanym kodzie 379
B.2.2. Przetwarzanie kodu źródłowego 379
B.2.3. Przetwarzanie kodu bajtowego 380
B.2.4. Introspekcja 380
B.2.5. Generowanie kodu bajtowego „w locie" 381
B.2.6. Obiekty „uogólnione" 382

C Powrót do świata rzeczywistego 383

C.1. Dziwna kopia 384
C.2. Im więcej, tym lepiej 385
C.3. Nie potrzebujemy kluczy głównych 385
C.4. Czas nie jest liniowy 386
C.5. Dynamicznie niebezpieczne 386
C.6. Synchronizować czy nie synchronizować? 387
C.7. Naprawdę gruby klient 388
C.8. Wznawianie Hibernate 388

Bibliografia 391

Skorowidz 393

2 Wprowadzenie i integracja Hibernate

W rozdziale:

- Działanie Hibernate na przykładzie aplikacji „Witaj świecie”
- Podstawowe interfejsy programistyczne Hibernate
- Integracja z zarządzanymi i niezarządzanymi środowiskami
- Zaawansowane opcje konfiguracyjne

Dobrze zdawać sobie sprawę z potrzeby odwzorowania obiektowo-relacyjnego w aplikacjach Javy, ale jeszcze lepiej zobaczyć takie odwzorowanie w akcji. Zacznijmy od prostego przykładu obrazującego zalety Hibernate.

Wiele osób wie, że niemal każda książka o programowaniu rozpoczyna się od przykładu „Witaj świecie". Nie zamierzamy się wyłamywać z tej konwencji i również przedstawimy prostą aplikację tego typu używającą Hibernate. Niestety, proste wyświetlenie komunikatu tekstowego na konsoli nie uwidoczniłoby żadnych zalet Hibernate. Z tego powodu tworzony program będzie tworzył nowe obiekty w bazie danych, aktualizował je i wykonywał zapytania, by je pobrać.

Niniejszy rozdział stanowi podstawę dla pozostałych rozdziałów. Poza przedstawieniem typowego dla książek informatycznych przykładu wprowadzimy również interfejs programistyczny Hibernate i omówimy konfigurację systemu w różnych środowiskach wykonawczych, na przykład serwerach aplikacji J2EE lub aplikacjach klienckich.

2.1. „Witaj świecie" w stylu Hibernate

Aplikacje Hibernate definiują **klasy trwałe**, które są odwzorowywane na tabele bazy danych. Pierwszy przykład zawiera jedną klasę i jeden plik opisu odwzorowania. Przyjrzyjmy się, jak wygląda prosta klasa trwałości, jak określa się odwzorowanie i jak wykonuje się podstawowe operacje dotyczące trwałości danych.

Celem pierwszej aplikacji jest zapamiętywanie w bazie danych komunikatów, a następnie ich pobieranie w celu wyświetlania. Aplikacja stosuje prostą klasę trwałości, `Message`, reprezentującą komunikaty do wyświetlenia. Klasę przedstawia listing 2.1.

Listing 2.1. Plik Message.java — prosta klasa trwałości

```
package hello;
public class Message {
    private Long id;
    private String text;
    private Message nextMessage;
    private Message() {}
    public Message(String text) {
        this.text = text;
    }
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    public String getText() {
        return text;
    }
    private void setText(String text) {
        this.text= text;
    }
```

Atrybut identyfikatora

Treść komunikatu

Referencja do innego komunikatu

```
public Long getNextMessage() {
    return nextMessage;
}
private void setnNextMessage(Long nextMessage) {
    this.nextMessage= nextMessage;
}
```

Klasa `Message` ma trzy atrybuty: identyfikator, treść komunikatu i referencję do innego komunikatu. Atrybut identyfikacyjny umożliwia aplikacji dostęp do identyfikatora bazodanowego — wartości klucza głównego — obiektu trwałego. Jeśli dwa egzemplarze klasy `Message` zawierają ten sam identyfikator, dotyczą tego samego wiersza w tabeli bazy danych. Jako typ identyfikatora wybraliśmy `Long`, choć nie jest to wymóg. Hibernate dopuszcza stosowane dla identyfikatorów dowolnych typów, co wkrótce przedstawimy.

Wszystkie atrybuty klasy `Message` stosują metody dostępowe do właściwości w stylu ziarenek JavaBeans. Klasa zawiera także bezparametrowy konstruktor. Klasy trwałości stosowane w kolejnych przykładach wyglądają bardzo podobnie do przedstawionej.

Egzemplarze klasy `Message` mogą być zarządzane (w sensie zapewnienia trwałości) przez Hibernate, ale nie jest to **przymus**. Ponieważ obiekt `Message` nie implementuje żadnych klas i interfejsów specyficznych dla Hibernate, można go używać jak dowolnej innej klasy Javy.

```
Message message = new Message("Witaj świecie");
System.out.println(message.getText());
```

Przedstawiony fragment wykonuje właśnie to zadanie, którego oczekuje się po aplikacji „Witaj świecie" — wyświetla na konsoli napis `Witaj świecie`. Niektórym wydaje się zapewne, że w tym momencie jesteśmy wyjątkowo uprzejmi — w rzeczywistości ten prosty przykład demonstruje istotną cechę wyróżniającą Hiberante od innych rozwiązań trwałości, na przykład ziarenek encyjnych EJB. Klasę trwałości można stosować w dowolnym kontekście, bo nie jest konieczny żaden szczególny kontener. Ponieważ książka dotyczy Hibernate, zapiszmy obiekt `Message` do bazy danych.

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
Message message = new Message("Witaj świecie");
session.save(message);
tx.commit();
session.close();
```

Przedstawiony kod korzysta z interfejsów `Session` i `Transaction` systemu Hibernate (wkrótce zajmiemy się wywołaniem `getSessionFactory()`). Przedstawiony kod skutkuje przekazaniem do bazy danych polecenia SQL podobnego do następującego:

```
insert into MESSSAGE (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID) values
(1, 'Witaj świecie', null)
```

Chwileczkę — kolumna MESSAGE_ID zostaje zainicjalizowana dziwną wartością. Nigdzie nie ustawialiśmy w kodzie właściwości id, więc oczekujemy dla niej wartości null, prawda? W rzeczywistości właściwość id jest szczególna — jest właściwością identyfikującą, która przechowuje wygenerowaną, unikatową wartość (w dalszej części tekstu wyjaśnimy sposób generowania wartości). Wartość zostaje wprowadzona do obiektu Message w momencie wykonania metody save().

W przykładzie zakładamy wcześniejsze istnienie tabeli MESSAGE. W rozdziale 9. omówimy wykorzystanie Hibernate do automatycznego tworzenia tabel wymaganych przez aplikację dzięki informacjom zawartym w plikach odwzorowań (w ten sposób unikamy pisania jeszcze większej ilości kodu SQL). Chcemy, by program wyświetlił komunikat na konsoli. Skoro komunikat znalazł się w bazie danych, nie powinno to sprawić żadnych trudności. Kolejny przykład pobiera z bazy danych wszystkie komunikaty w kolejności alfabetycznej i wyświetla ich zawartość.

```
Session newSession = getSessionFactory().openSession();
Transaction newTransaction = newSession.beginTransaction();
List messages = newSession.find("from Message as m order by m.text asc");
System.out.println("Znalezionych komunikatów: " + messages.size());
for (Iterator iter = messages.iterator(); iter.hasNext();) {
    Message message = (Message) iter.next();
    System.out.println(message.getText());
}
newTransaction.commit();
newSession.close();
```

Ciąg znaków "from Message as m order by m.text asc" to zapytanie Hibernate zapisane w specjalnym, obiektowym języku zapytań Hibernate (HQL — *Hibernate Query Language*). Zapytanie zostaje wewnętrznie przekształcone na następujący kod SQL w momencie wywołania metody find().

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
order by m.MESSAGE_TEXT asc
```

Przedstawiony kod powoduje wyświetlenie na konsoli poniższego tekstu:

```
Znalezionych komunikatów: 1
Witaj świecie
```

Jeśli ktoś nigdy wcześniej nie korzystał z narzędzi ORM, spodziewa się ujrzeć kod SQL w metadanych lub też kodzie Javy. Nie ma go tam. Cały kod SQL zostaje wygenerowany dynamicznie w trakcie działania aplikacji (ewentualnie w momencie jej uruchamiania dla wszystkich instrukcji SQL wielokrotnego użytku).

Aby wystąpiła cała magia, Hibernate potrzebuje więcej informacji na temat zapewniania trwałości klasie Message. Informację tę najczęściej umieszcza się **dokumencie odwzorowania** zapisywanym w formacie XML. Dokument ten definiuje, poza innymi kwestiami, w jaki sposób przełożyć właściwości klasy Message na kolumny tabeli MESSAGES. Przyjrzyjmy się fragmentowi dokumentu odwzorowania — listing 2.2.

Listing 2.2. Prosty plik odwzorowania Hibernate w formacie XML

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">
<hibernate-mapping>
    <class
        name="hello.Message"
        table="MESSAGES">
        <id
            name="id"
            column="MESSAGE_ID">
            <generator class="increment"/>
        </id>
        <property
            name="text"
            column="MESSAGE_TEXT"/>
        <many-to-one
            name="nextMessage"
            cascade="all"
            column="NEXT_MESSAGE_ID"/>
    </class>
</hibernate-mapping>
```

Warto zauważyć, że Hibernate 2.0 i 2.1 stosują ten sam schemat DTD!

Dokument odwzorowania informuje Hibernate, że klasa `Message` przekłada się na tabelę `MESSAGES`. Informuje również, że właściwość identyfikatora ma trafić do kolumny nazwanej `MESSAGE_ID`, właściwość komunikatu do kolumny o nazwie `MESSAGE_TEXT`, natomiast właściwość `nextMessage` jest asocjacją z **krotnością wiele-do-jednego** i powinna trafić do kolumny o nazwie `NEXT_MESSAGE_ID` (pozostałymi szczegółami na tym etapie nie będziemy się zajmować).

Dokument XML nie jest trudny do zrozumienia. Można go bez problemów ręcznie tworzyć i modyfikować. W rozdziale 3. omówimy sposób generowania tego pliku na podstawie komentarzy zawartych w kodzie źródłowym aplikacji. Niezależnie od wybranego podejścia Hiberante uzyskuje wystarczająco dużo informacji, by poprawnie generować wszystkie polecenia SQL dotyczące wstawiania, aktualizacji, usuwania i pobierania egzemplarzy klasy `Message`. Nie trzeba ręcznie pisać wszystkich wymaganych poleceń SQL.

Uwaga Wielu programistów Javy narzeka na tak zwane „piekło metadanych", które towarzyszy tworzeniu oprogramowania J2EE. Niektórzy sugerują nawet rezygnację z metadanych XML i powrót do zwykłego kodu Javy. Choć w pełni popieramy te dążenia w niektórych dziedzinach, ORM wydaje się być zagadnieniem, w którym metadane naprawdę są potrzebne. Hibernate stosuje sensowne wartości domyślne, by zminimalizować długość plików konfiguracyjnych. Wykorzystuje dojrzałą definicję typu dokumentu (DTD), więc możliwe jest wykorzystanie edytorów z uzupełnianiem składni i walidacją. Istnieją nawet narzędzia zapewniające automatyczne generowanie metadanych.

Zmieńmy pierwszy komunikat i dodatkowo utwórzmy kolejny powiązany z pierwszym. Kod wykonujący to zadanie przedstawia listing 2.3.

Listing 2.3. Aktualizacja komunikatu

```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();

// 1 to identyfikator wygenerowany dla pierwszego komunikatu
Message message = (Message) session.load(Message.class, new Long(1));
message.setText("Witajcie Ziemianie");
Message nextMessage = new Message("Proszę zabrać mnie do waszego przywódcy");
message.setNextMessage(nextMessage);
tx.commit();
session.close();
```

Powyższy kod powoduje wykonanie w jednej transakcji trzech poleceń SQL.

```
select m.MESSAGE_ID, m.MESSAGE_TEXT, m.NEXT_MESSAGE_ID
from MESSAGES m
where m.MESSAGE_ID = 1

insert into MESSAGES (MESSAGE_ID, MESSAGE_TEXT, NEXT_MESSAGE_ID)
values (2, 'Proszę zabrać mnie do waszego przywódcy', null)

update MESSAGES
set MESSAGE_TEXT = 'Witajcie Ziemianie', NEXT_MESSAGE_ID = 2
where MESSAGE_ID = 1
```

Zauważ, że Hibernate automatycznie rozpoznał modyfikację właściwości `text` i `nextMessage` dla pierwszego komunikatu i zapisał nowe wartości w bazie danych. Skorzystaliśmy z cechy Hibernate nazywanej **automatycznym wykrywaniem zabrudzenia**. Pozwala ona uniknąć jawnego proszenia Hibernate o aktualizację bazy danych, gdy zmienia się stan obiektu wewnątrz transakcji. Warto zwrócić uwagę, iż nowy komunikat również stał się trwały, gdy tylko powstała referencja do niego z pierwszego komunikatu. Jest to tak zwany zapis kaskadowy — unika się w ten sposób jawnego wywoływania metody `save()` dla nowego obiektu trwałego, o ile tylko jest on osiągalny z poziomu innego obiektu trwałego. Kolejność wykonania instrukcji SQL nie jest taka sama jak kolejność modyfikacji wartości właściwości. Hibernate używa wyrafinowanego algorytmu do określenia wydajnej kolejności zapisów, by uniknąć zerwania ograniczeń klucza obcego i jednocześnie pozostać odpowiednio przewidywalnym. Jest to tak zwany **transakcyjny zapis opóźniony**.

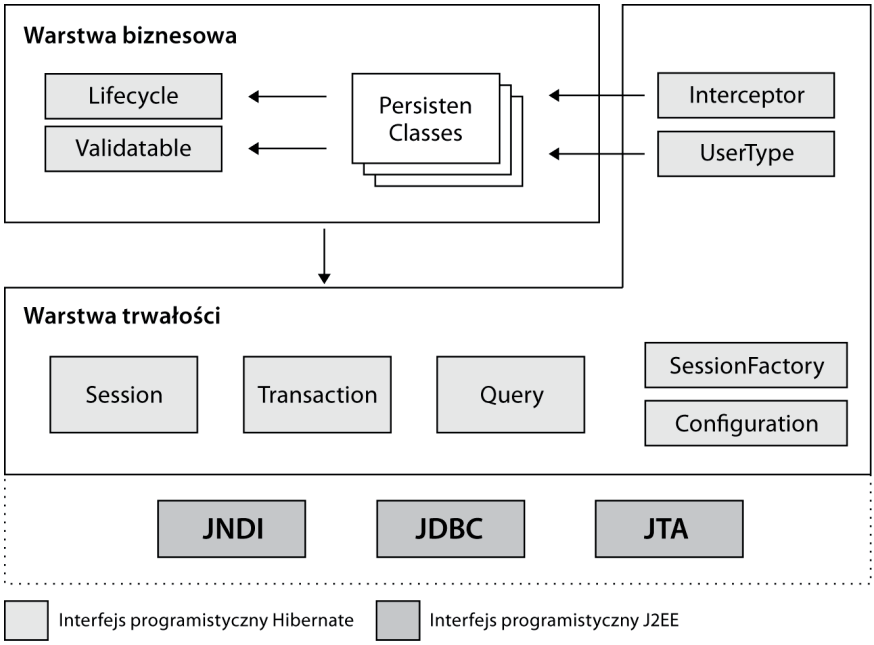
Ponownie uruchomienie programu spowoduje wyświetlenie poniższego wyniku:

```
Znalezionych komunikatów: 2
Witajcie Ziemianie
Proszę zabrać mnie do waszego przywódcy
```

Na tym zakończymy przygodę z aplikacją „Witaj świecie". Gdy poznaliśmy działający kod, możemy się cofnąć i dokładniej omówić poszczególne elementy głównego interfejsu programistycznego Hibernate.

2.2. Podstawy architektury Hibernate

Interfejs programistyczny to pierwszy element Hibernate, który trzeba poznać, by móc skorzystać z warstwy trwałości w aplikacji. Głównym celem projektowym interfejsu była minimalizacja współzależności między poszczególnymi komponentami oprogramowania. W praktyce interfejs programistyczny ORM nie należy do najmniejszych. Na szczęście nie wszystkie interfejsy trzeba poznać od razu, by korzystać z Hibernate. Rysunek 2.1 przedstawia zadania najważniejszych interfejsów w warstwie biznesowej i trwałości. Na rysunku warstwa biznesowa znajduje się powyżej warstwy trwałości, gdyż w typowych aplikacjach warstwowych działa jako klient warstwy trwałości. Niektóre proste aplikacje niezbyt dobrze separują logikę biznesową od trwałości — nie będziemy się tym jednak zajmować, bo zalecane podejście przedstawia rysunek.



Rysunek 2.1. Wysokopoziomowy przegląd interfejsów programistycznych Hibernate z uwzględnieniem architektury warstwowej

Interfejsy Hibernate przedstawione na rysunku 2.1 można podzielić następująco:

- Interfejsy wywoływane przez aplikacje w celu wykonania prostych operacji CRUD i wyszukiwania. Interfejsy te stanowią główny punkt styku logiki biznesowej z Hibernate. Zawierają interfejsy: `Session`, `Transaction` i `Query`.
- Interfejsy wywoływane przez kod infrastruktury aplikacji w celu konfiguracji Hibernate. Jest to przede wszystkim klasa `Configuration`.
- Interfejsy wywołań zwrotnych pozwalające aplikacji reagować na zdarzenia zachodzące wewnątrz Hibernate. Są to interfejsy: `Interceptor`, `Lifecycle` i `Validatable`.
- Interfejsy zapewniające rozszerzanie rozbudowanej funkcjonalności odwzorowania w Hibernate. Są to interfejsy: `UserType`, `CompositeUserType` i `IdentifierGenerator`. Interfejsy te implementuje kod infrastruktury aplikacji (jeśli to konieczne).

Hibernate wykorzystuje istniejące interfejsy programistyczne Javy, włączając w to JDBC, JTA (*Java Transaction API*) oraz JNDI (*Java Naming and Directory Interface*). JDBC zapewnia odpowiedni poziom abstrakcji funkcji wspólnych dla różnych systemów relacyjnych baz danych, co oznacza, że Hibernate potrafi skorzystać z dowolnej bazy danych ze sterownikiem JDBC. JNDI i JTA umożliwiają Hibernate integrację z serwerami aplikacji J2EE.

W niniejszym podrozdziale nie będziemy omawiać szczegółowej semantyki metod interfejsów Hibernate. Zajmiemy się raczej podstawowymi rolami interfejsów. Większość interfejsów znajduje się w pakiecie `net.sf.hibernate`. Przyjrzyjmy się pokrótce każdemu z nich.

2.2.1. Interfejsy podstawowe

Pięć interfejsów podstawowych pojawia się w niemal każdej aplikacji wykorzystującej Hibernate. Dzięki nim możliwe staje się pobieranie i zapamiętywanie trwałych obiektów oraz sterowanie transakcjami.

Interfejs Session

Interfejs `Session` to główny interfejs każdej aplikacji Hibernate. Egzemplarze `Session` są lekkie — koszt ich utworzenia i zniszczenia bywa niewielki. Jest to niezwykle istotne, bo każda aplikacja cały czas tworzy i usuwa coraz to nowe sesje, prawdopodobnie przy każdym nowym żądaniu. Sesje Hibernate **nie** są bezpieczne wielowątkowo i z tego powodu należy tak projektować aplikację, by używać ich w danym momencie w tylko jednym wątku.

Znaczenie **sesji** Hibernate można opisać jako coś pomiędzy **połączeniem** i **transakcją**. Najprościej traktować sesję jako bufor lub kolekcję załadowanych obiektów powiązanych z jedną jednostką zadaniową. Hibernate potrafi wykryć zmiany w obiektach tej jednostki zadaniowej. Często interfejs `Session` nazywa się zarządcą trwałości, gdyż zapewnia dostęp do podstawowych operacji trwałości takich jak zapis i pobieranie obiektów. Warto podkreślić, że sesja Hibernate nie ma nic wspólnego z sesją warstwy połączenia internetowego, na przykład

z `HttpSession`. Gdy w książce używamy słowa sesja, mamy na myśli sesję Hibernate. Gdy mówimy o sesji użytkownika, mamy na myśli obiekt `HttpSession`.

Interfejs `Session` został dokładnie omówiony w podrozdziale 4.2.

Interfejs SessionFactory

Aplikacja pobiera egzemplarze `Session` z `SessionFactory`. W porównaniu z `Session` obiekt ten jest znacznie bardziej interesujący.

Interfejs `SessionFactory` pewnością nie jest lekki! Został zaprojektowany z myślą o współdzieleniu przez wiele wątków aplikacji. Najczęściej na całą aplikację występuje tylko jeden obiekt `SessionFactory` tworzony w momencie jej inicjalizacji. Jeśli jednak aplikacja korzysta z kilku baz danych w połączeniu z Hibernate, potrzeba osobnego obiektu `SessionFactory` dla każdej bazy danych.

Obiekt `SessionFactory` buforuje wygenerowane polecenia SQL i inne metadane odwzorowania stosowane przez Hibernate w trakcie działania aplikacji. Dodatkowo buforuje dane, które zostały odczytane w jednej jednostce zadaniowej i mogłyby przydać się w innej jednostce zadaniowej (ten **drugi poziom buforowania** działa tylko wtedy, gdy odwzorowania klas i kolekcji sugerują jego użycie).

Interfejs Configuration

Obiekt `Configuration` służy do konfiguracji i uruchomienia Hibernate. Aplikacja używa egzemplarza `Configuration` do wskazania położenia dokumentów odwzorowań i właściwości specyficznych dla Hibernate. Podrozdział 2.3 ze średnim stopniem złożoności opisuje konfigurację systemu Hibernate.

Interfejs Transaction

Interfejs `Transaction` jest opcjonalny. Aplikacje Hibernate nie muszą z niego korzystać, jeśli chcą zarządzać transakcjami we własnym zakresie. Interfejs stara się ukryć szczegóły implementacji konkretnych mechanizmów transakcyjnych: JDBC, klasy `UserTransaction` z JTA lub nawet transakcji CORBA (*Common Object Request Broker Architecture*). W ten sposób zapewnia aplikacji jednorodny sposób ich obsługi. Ułatwia to zachowanie przenośności aplikacji Hibernate między różnymi środowiskami wykonywania i kontenerami.

W niniejszej książce stosujemy interfejs `Transaction`. Został on dokładnie wyjaśniony w rozdziale 5.

Interfejsy Query i Criteria

Interfejs `Query` umożliwia wysyłanie zapytań do bazy danych i sterowanie procesem ich wykonywania. Zapytania pisze się w języku HQL lub też w odpowiednim dla danej bazy danych dialekcie SQL. Egzemplarz `Query` odpowiada za dowiązanie parametrów zapytania, ograniczenie liczby zwracanych wyników i wykonanie zapytania.

Interfejs `Criteria` jest bardzo podobny. Umożliwia utworzenie i wykonanie obiektowych kryteriów wyszukiwania.

Aby tworzony kod był możliwie krótki, Hibernate dostarcza kilka skrótów w interfejsie `Session`, by móc w jednym wierszu wykonać zapytanie. W książce nie stosujemy tych skrótów — zawsze używamy interfejsu `Query`.

Egzemplarz `Query` jest lekki nie daje się zastosować poza egzemplarzem `Session`, dla którego powstał. Opis funkcji interfejsu `Query` znajduje się w rozdziale 7.

2.2.2. Interfejsy wywołań zwrotnych

Interfejsy wywołań zwrotnych umożliwiają aplikacji otrzymywanie powiadomień, gdy coś ciekawego stanie się ze sprawdzanym obiektem — na przykład zostanie załadowany, zapisany lub usunięty. Aplikacje stosujące Hibernate nie muszą korzystać z wywołań zwrotnych, ale niejednokrotnie przydają się one do tworzenia ogólnej funkcjonalności, na przykład automatycznego tworzenia zapisów audytorskich.

Interfejsy `Lifecycle` i `Validatable` zapewniają trwałym obiektom reakcję na zdarzenia związane z własnym **cyklem życia trwałego**. Cykl życia ma nierozerwalny związek z operacjami CRUD obiektu. Zespół projektowy Hibernate był pod silnym wpływem innych rozwiązań ORM posiadających podobne wywołania zwrotne. Później zdano sobie sprawę, że kasy trwałości implementujące interfejsy specyficzne dla Hibernate nie są najlepszym pomysłem, gdyż ich stosowanie utrudnia przenośność kodu. Ponieważ ich używanie obecnie nie jest zalecane, nie są one szerzej opisywane w niniejszej książce.

Interfejs `Interceptor` wprowadzono, by aplikacje mogły przetwarzać wywołania zwrotne bez potrzeby wprowadzania implementacji interfejsów Hibernate do klas trwałości. Implementacje interfejsu `Interceptor` wprowadza się do egzemplarzy trwałych klas jako parametry. Przykład ich użycia pojawi się w rozdziale 8.

2.2.3. Typy

Podstawowym i niezwykle użytecznym elementem architektury jest sposób wykonania obiektu `Type` w Hibernate. Obiekt ten odwzorowuje typ Javy na typ kolumny tabeli bazy danych (w rzeczywistości typ może dotyczyć wielu kolumn). Wszystkie trwałe właściwości klas trwałych, włączając w to asocjacje, posiadają odpowiedni typ Hibernate. Ten sposób obsługi czyni Hibernate wyjątkowo elastycznym i rozszerzalnym.

Istnieje bogaty zbiór typów wbudowanych zawierający wszystkie typy podstawowej Javy oraz wiele klas ze standardowej biblioteki JDK, włączając w to typy: `java.util.Currency`, `java.util.Calendar`, `byte[]` i `java.io.Serializable`.

Co więcej, Hibernate obsługuje **typy zdefiniowane przez użytkownika**. Interfejsy `UserType` i `CompositeUserType` dają możliwość kreowania własnych typów. W ten sposób klasy wykorzystywane w wielu różnych aplikacjach, na przykład `Address`, `Name` i `MonetaryAmount` udaje się obsłużyć wygodnie i elegancko. Własne typy uważa się za jedną z najistotniejszych cech Hibernate. Zalecamy z nich korzystać w coraz to nowy sposób!

Szczegóły dotyczące typów Hibernate zostały omówione w podrozdziale 6.1.

2.2.4. Interfejsy rozszerzeń

Większość funkcjonalności Hibernate można dostosować do własnych potrzeb, włączając w to wybór wbudowanych strategii. Gdy okazują się one niewystarczające, Hibernate dopuszcza zastosowanie własnej implementacji przez zastosowanie odpowiedniego interfejsu. Możliwe rozszerzenia są następujące:

- generacja klucza głównego (interfejs `IdentifierGenerator`),
- obsługa dialektu języka SQL (klasa abstrakcyjna `Dialect`),
- strategia buforowania (interfejsy `Cache` i `CacheProvider`),
- zarządzanie połączeniem JDBC (interfejs `ConnectionProvider`),
- zarządzanie transakcjami (interfejsy `TransactionFactory`, `Transaction` i `TransactionManagerLookup`),
- strategia ORM (hierarchia interfejsów `ClassPersister`),
- strategia dostępu do właściwości (interfejs `PropertyAccessor`),
- tworzenie pośredników (interfejs `ProxyFactory`).

Hibernate zawiera co najmniej jedną implementację każdego z wymienionych interfejsów, więc najczęściej nie trzeba zaczynać od podstaw, gdy chce się jedynie rozszerzyć pewną wbudowaną funkcjonalność. Co więcej, dostępny kod źródłowy stanowi doskonały przykład w sytuacji, gdy tworzy się własne rozwiązanie.

Zanim zacznie się tworzyć jakikolwiek kod związany z Hibernate, warto zadać sobie następujące pytanie: w jaki sposób zmusić do działania obiekt `Session`?

2.3. Konfiguracja podstawowa

Przyjrzeliśmy się przykładowej aplikacji oraz podstawowym interfejsom Hibernate. Aby móc skorzystać z automatycznej trwałości, trzeba poznać sposób jej konfiguracji. Hibernate udaje się tak skonfigurować, by działało poprawnie w niemalże dowolnej aplikacji Javy i środowisku programistycznym. Na ogół jednak stosuje się go w dwu- lub trójwarstowych aplikacjach klient-serwer, przy czym Hibernate zostaje wdrożony tylko po stronie serwera. Aplikacją kliencką najczęściej jest przeglądarka internetowa, ale zdarzają się również aplikacje SWT i Swing. Choć w książce skupiamy się na wielowarstwowych aplikacjach internetowych, przedstawiane opisy mają zastosowanie również w dowolnej innej architekturze, na przykład aplikacjach wiersza poleceń. Istotnym jest, by zrozumieć różnicę w konfiguracji Hibernate w środowisku zarządzanym i niezarządzanym.

- **Środowisko zarządzanie** — tworzy pule zasobów, na przykład połączeń z bazą danych, i dopuszcza deklaratywne (w postaci metadanych) określanie zakresu transakcji oraz zasad bezpieczeństwa. Serwery aplikacji J2EE, na przykład JBoss, BEA WebLogic lub IBM WebSphere, implementują w Javie standardowe (zgodne z J2EE) środowisko zarządzane.
- **Środowisko niezarządzane** — zapewnia proste zarządzanie współbieżnością dzięki puli wątków. Pojemnik serwletowy taki jak Jetty lub Tomcat

zapewnia niezarządzane środowisko serwerowe dla aplikacji internetowych pisanych w Javie. Samowystarczalne aplikacje kliencie i aplikacje wiersza poleceń również uważa się za niezarządzane. Środowiska te nie udostępniają automatycznego zarządzania transakcjami i zasobami oraz nie zapewniają infrastruktury bezpieczeństwa. Sama aplikacja zarządza połączeniami z bazą danych i określa granice transakcji.

Hibernate stara się w sposób abstrakcyjny traktować środowisko, w którym przychodzi mu pracować. W środowisku niezarządzanym sam obsługuje transakcje i połączenia JDBC (lub deleguje ich wykonanie do odpowiedniego kodu aplikacji). W środowisku zarządzanym integruje się z transakcjami i połączeniami bazodanowymi zarządzanymi przez kontener. Hibernate potrafi poprawnie działać w obu środowiskach.

W obu środowiskach pierwszym zadaniem jest uruchomienie Hibernate. W praktyce nie jest to trudne — wystarczy utworzyć obiekt `SessionFactory`, używając klasy `Connection`.

2.3.1. Tworzenie obiektu SessionFactory

Aby uzyskać obiekt `SessionFactory`, należy najpierw utworzyć pojedynczy egzemplarz klasy `Configuration` w momencie inicjalizacji aplikacji i skonfigurować ścieżki do plików odwzorowań. Po konfiguracji obiekt `Configuration` pozwala tworzyć egzemplarze `SessionFactory`. Po ich wykonaniu obiekt `Configuration` staje się zbędny.

Poniższy kod uruchamia Hibernate:

```
Configuration cfg = new Configuration();
cfg.addResource("hello/Message.hbm.xml");
cfg.setProperties(System.getProperties());
SessionFactory sessions = cfg.buildSessionFactory();
```

Lokalizację pliku odwzorowania, *Message.hbm.xml*, określa się względem podstawowej ścieżki klas aplikacji. Jeśli na przykład ścieżka klas jest aktualnym folderem, plik *Message.hbm.xml* musi znajdować się wewnątrz folderu *hello*. Pliki odwzorowania XML muszą znajdować się w ścieżce wyszukiwania klas. W przykładzie wykorzystujemy również właściwości systemowe maszyny wirtualnej, by ustawić pozostałe opcje konfiguracyjne (można je inicjalizować wcześniej w samej aplikacji lub przy uruchamianiu systemu maszyny wirtualnej).

Łańcuch wywołań metod

Tworzenie łańcuchów wywołań metod to styl programowania obsługiwany przez wiele interfejsów Hibernate. Jest on bardziej popularny w języku Smalltalk niż w Javie. Niektóre osoby uważają go za mniej czytelny i trudniejszy do debugowania niż typowy styl programowania w Javie. W większości sytuacji jest jednak wyjątkowo wygodny.

Większość programistów Javy tworzy metody ustawiające lub dodające zwracające typ `void`, czyli brak wartości. Język SmallTalk nie posiada typy `void`, więc metody ustawiające lub dodające zwracają obiekt otrzymujący dane. Takie podejście umożliwia zmianę wcześniejszego kodu na następujący.

```
SessionFactory sessions = new Configuration()
    .addResource("hello/Message.hbm.xml")
    .setProperties(System.getProperties())
    .buildSessionFactory();
```

Zauważ, że nie potrzeba deklarować lokalnej zmiennej dla obiektu `Configuration`. Styl ten będzie stosowany w niektórych przykładach, ale gdy ktoś go nie lubi, nie wymuszamy jego stosowania. Jeśli ktoś go **lubi**, zalecamy pisanie każdego wywołania metody w osobnym wierszu. W przeciwnym razie znacznie trudniej testować utworzony kod debugerem.

Przyjęło się, że pliki odwzorowań Hibernate w formacie XML zapisuje się w plikach z rozszerzeniem *.hbm.xml*. Dodatkowo przyjęło się, by tworzyć osobny plik odwzorowania dla każdej klasy, zamiast umieszczać cały opis odwzorowań w jednym pliku (jest to możliwe, ale nie jest zalecane). Przedstawiony przykład „Witaj świecie" stosuje tylko jedną klasę trwałości, ale załóżmy, że jest ich wiele i każde odwzorowanie znajduje się w osobnym pliku. Gdzie należy umieścić pliki odwzorowań?

Dokumentacja Hibernate zaleca, by plik odwzorowania dla każdej klasy trwałości umieszczać w tym samym folderze co klasę. Przykładowo, plik odwzorowania dla klasy `Message` należy umieścić w folderze *hello* pod nazwą *Message.hbm.xml*. Dla innej klasy trwałej należałoby wykonać osobny plik odwzorowania. Zalecamy stosować się do przedstawionej sugestii. Monolityczne pliki metadanych narzucane przez niektóre systemy szkieletowe, na przykład *struts-config.xml* z Struts, są głównym czynnikiem powodującym powstanie hasła „piekło metadanych". Wiele plików odwzorowań wczytuje się, wielokrotnie wywołując metodę `addResource()`. Ewentualnie, gdy programista stosuje się do przedstawionej powyżej konwencji, może użyć metody `addClass()`, przekazując jako parametr klasę trwałą.

```
SessionFactory sessions = new Configuration()
.addClass(org.hibernate.auction.model.Item.class)
.addClass(org.hibernate.auction.model.Category.class)
.addClass(org.hibernate.auction.model.Bid.class)
.setProperties(System.getProperties())
.buildSessionFactory();
```

Metoda `addClass()` zakłada, że nazwa pliku odwzorowania kończy się rozszerzeniem *.hbm.xml* i znajduje się w tym samym folderze co odwzorowywany plik klasy.

Przedstawimy tworzenie pojedynczego obiektu `SessionFactory`, ponieważ takie podejście występuje w aplikacjach najczęściej. Jeśli potrzebny jest kolejny obiekt, bo na przykład istnieje kilka baz danych, cały proces powtarza się od nowa. W ten sposób zawsze istnieje jeden obiekt `SessionFactory` na bazę danych gotowy do tworzenia obiektów Session działających z tą bazą danych i zestawem odwzorowań klas.

Konfiguracja Hibernate to nie tylko wskazanie dokumentów odwzorowań. Trzeba również wskazać sposób uzyskania połączenia z bazą danych oraz określić zachowanie Hibernate w różnych sytuacjach. Mnogość dostępnych właściwości konfiguracyjnych potrafi przytłoczyć (pełna ich lista znajduje się w dokumentacji

Hibernate). Na szczęście większość parametrów stosuje sensowne wartości domyślne, więc najczęściej potrzeba zmienić tylko ich niewielką część.

Aby określić opcje konfiguracyjne, korzysta się z jednej z wymienionych technik.

- Przekazuje się egzemplarz klasy `java.util.Properties` jako parametr metody `Configuration.setProperties()`.
- Ustawia się właściwości systemowe za pomocą konstrukcji `java -Dwłaściwość=wartość`.
- Umieszcza się w ścieżce wyszukiwania klas plik o nazwie *hibernate.properties*.
- Dołącza się elementy `<property>` do pliku *hibernate.cfg.xml* znajdującego się w ścieżce wyszukiwania klas.

Techniki pierwszą i drugą stosuje się naprawdę rzadko — jedynie w trakcie testów lub szybkiego prototypowania. Większość aplikacji potrzebuje stałego pliku konfiguracyjnego. Oba pliki, *hibernate.properties* i *hibernate.cfg.xml*, pełną tę samą funkcję — konfigurują Hibernate. Wybór pliku zależy tak naprawdę od własnych preferencji co do składni właściwości. Możliwe jest nawet mieszanie obu technik i posiadanie osobnych ustawień dla środowiska testowego i wdrożeniowego. Przedstawimy to zagadnienie w dalszej części rozdziału.

Bardzo rzadko stosowanym rozwiązaniem alternatywnym jest przekazywanie przez aplikację obiektu `Connection` z JDBC w momencie tworzenia obiektu `Session` dzięki obiektowi `SessionFactory` (powstaje wtedy kod typu `sessions.openSession(myConnection)`). Opcja ta pozwala nie podawać w trakcie konfiguracji żadnych opcji związanych z połączeniem z bazą danych. Nie polecamy takiego podejścia dla nowych aplikacji, które mogą skorzystać z infrastruktury połączeń bazodanowych całego środowiska (na przykład wykorzystując pulę połączeń JDBC lub źródła danych serwera aplikacji).

Ze wszystkich opcji konfiguracyjnych najważniejsze są ustawienia połączenia bazodanowego. Różnią się w zależności od tego, czy stosuje się środowisko zarządzane czy niezarządzane. Z tego względu opis rozbijemy na dwa przypadki. Zacznijmy od środowiska niezarządzanego.

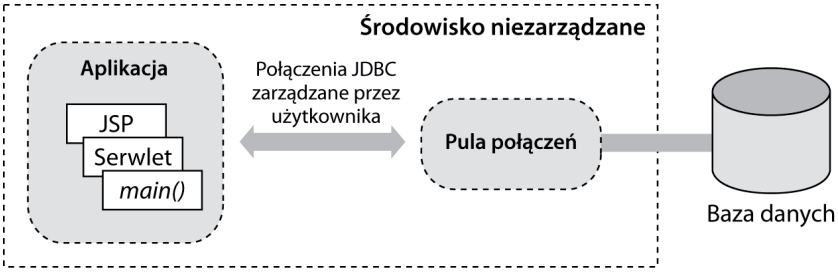
2.3.2. Konfiguracja w środowisku niezarządzanym

W środowisku niezarządzanym, na przykład kontenerze serwletów, aplikacja odpowiada za uzyskanie połączeń JDBC. Hibernate jest częścią aplikacji, więc również może uczestniczyć w ich uzyskaniu. Plik konfiguracyjny informuje, w jaki sposób Hibernate może uzyskać (lub utworzyć nowe) połączenia JDBC. Ogólnie nie zaleca się tworzenia połączeń za każdym razem, gdy tylko chce się skorzystać z bazy danych. Aplikacja Javy powinna zatem używać puli połączeń JDBC. Istnieją trzy powody przemawiające ze stosowaniem puli:

- Uzyskiwanie nowych połączeń jest kosztowne.
- Utrzymywanie wielu niewykorzystywanych połączeń jest kosztowne.

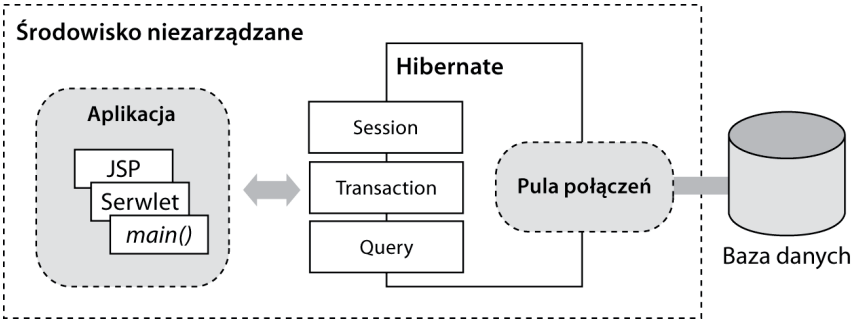
- Tworzenie instrukcji przygotowanych dla niektórych sterowników również jest kosztowne.

Rysunek 2.2 przedstawia rolę puli połączeń JDBC w środowisku wykonawczym aplikacji internetowej. Ponieważ samo środowisko nie udostępnia puli połączeń bazodanowych, aplikacja musi wprowadzić własny algorytm tworzenia puli lub stosować w tym celu niezależną bibliotekę, na przykład udostępnianą na zasadach *open source* bibliotekę C3P0. Aplikacje bez Hibernate najczęściej wywołuje metody puli połączeń, by uzyskać połączenie JDBC i wykonać polecenie SQL.



Rysunek 2.2. Pula połączeń JDBC w środowisku niezarządzanym

Po dodaniu Hibernate schemat ulega zmianie — ORM działa jako klient puli połączeń JDBC, co przedstawia rysunek 2.3. Kod aplikacji używa interfejsów programistycznych `Session` i `Query` z Hibernate do wykonywania operacji trwałości danych. Zajmuje się również zarządzaniem transakcjami, używając interfejsu `Transaction` z Hibernate.



Rysunek 2.3. Hibernate z pulą połączeń w środowisku niezarządzanym

Wykorzystanie puli połączeń

Hibernate stosuje architekturę modułów rozszerzających, więc potrafi zintegrować się z dowolną pula połączeń. Obsługa C3P0 została wbudowana, więc z niej skorzystamy. Hibernate same utworzy pulę po podaniu wymaganych parametrów. Przykład pliku `hibernate.properties` stosującego C3P0 przedstawia listing 2.4.

Listing 2.4. Plik hibernate.properties z ustawieniami puli połączeń C3P0

```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/auctiondb
hibernate.connection.username = auctionuser
hibernate.connection.password = secret
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=300
hibernate.c3p0.max_statements=50
hibernate.c3p0.idle_test_period=3000
```

Przedstawiony fragment pliku określa następujące informacje omawiane wiersz po wierszu:

- Nazwę klasy Javy implementującej sterownik JDBC dla konkretnej bazy danych (klasa `Driver`). Plik JAR sterownika musi znajdować w ścieżce wyszukiwania klas aplikacji.
- Adres URL w formacie JDBC określający adres serwera i nazwę bazy danych dla połączeń JDBC.
- Nazwa użytkownika bazy danych.
- Hasło do bazy danych dla wskazanego użytkownika.
- Klasa dialektu języka SQL. Pomimo wysiłków standaryzacyjnych organizacji ANSI język SQL został zaimplementowany inaczej przez poszczególnych twórców baz danych. Należy więc określić klasę Dialekt dla stosowanych połączeń. Hibernate zawiera wbudowaną obsługę najpopularniejszych baz danych SQL. Nowe dialekty można łatwo dodawać.
- Minimalna liczba oczekujących na działania połączeń JDBC utrzymywana przez C3P0.
- Maksymalna liczba połączeń w puli. Zostanie zgłoszony wyjątek wykonania, gdy wartość ta okaże się niewystarczająca.
- Okres bezczynności podawany w sekundach, więc w przykładzie jest to 5 minut. Po tym okresie nieużywane połączenie zostaje usunięte z puli.
- Maksymalna liczba zbuforowanych poleceń przygotowanych. Buforowanie poleceń przygotowanych pozwala znacząco zwiększyć szybkość działania Hibernate.
- Czas bezczynności w sekundach, po którym połączenie jest automatycznie poddawane walidacji.

Określanie właściwości w postaci `hinernate.c3p0.*` powoduje automatyczne wybranie C3P0 jako rozwiązania puli połączeń dla Hibernate (nie potrzeba żadnej dodatkowej opcji w celu włączenia C3P0). Biblioteka C3P0 ma znacznie więcej funkcji i opcji niż zostało przedstawionych w przykładzie. Zalecamy zajrzeć do dokumentacji interfejsu programistycznego Hibernate. Dokumentacja dla klasy

`net.sf.hibernate.cfg.Environment` informuje o wszystkich właściwościach konfiguracyjnych, włączając w to opcje C3P0 i innych puli połączeń obsługiwanych w sposób bezpośredni przez Hibernate.

Pozostałymi obsługiwanymi pulami połączeń są: Apache DBCP i Proxool. Warto wypróbować we własnym środowisku wszystkie trzy rozwiązania przed podjęciem decyzji. Społeczność Hibernate najczęściej wybiera C3P0 i Proxool.

Hibernate zawiera dodatkowo własny mechanizm puli połączeń, ale nadaje się on jedynie do testów i eksperymentów z Hibernate — **nie** należy go stosować w systemie produkcyjnym. Nie był projektowany z uwzględnieniem wydajnej obsługi wielu współbieżnych żądań. Brakuje mu również funkcji wykrywania niepowodzeń istniejących w wyspecjalizowanych pulach połączeń.

Uruchamianie Hibernate

W jaki sposób uruchomić Hibernate z przedstawionymi właściwościami? Jeśli właściwości zadeklaruje się w pliku *hibernate.properties*, wystarczy umieścić ten plik w ścieżce wyszukiwania klas aplikacji. Zostanie automatycznie wykryty i wczytany przez Hibernate w momencie tworzenia obiektu `Configuration`.

Podsumujmy poznane do tej pory kroki konfiguracyjne. Najwyższy czas pobrać i zainstalować Hibernate, jeśli chce się go użyć w środowisku niezarządzanym.

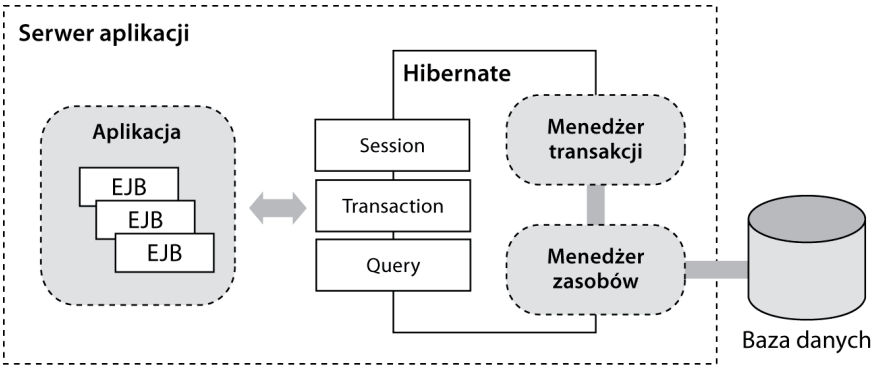
1. Pobierz i rozpakuj sterownik JDBC dla bazy danych. Najczęściej jest dostępny na witrynie producenta bazy danych. Umieść plik JAR w ścieżce wyszukiwania klas aplikacji. To samo zrób z plikiem JAR Hibernate.
2. Dodaj zależności Hibernate do ścieżki wyszukiwania klas. Są one dołączone do Hibernate w folderze *lib*. Warto zajrzeć do pliku *lib/README.txt*, by poznać listę wymaganych i opcjonalnych bibliotek.
3. Wybierz pulę połączeń JDBC obsługiwaną przez Hibernate i skonfiguruj ją, używając pliku właściwości. Nie zapomnij o wskazaniu dialektu języka SQL.
4. Niech klasa `Configuration` potrafi odnaleźć plik właściwości. Nadaj mu nazwę *hibernate.properties* i umieść go w ścieżce wyszukiwania klas.
5. Utwórz egzemplarz klasy `Configuration` w aplikacji i załaduj pliki odwzorowania w formacie XML, używając metod `addResource()` lub `addClass()`. Utwórz obiekt `SessionFactory` na podstawie `Configuration`, wywołując metodę `buildSessionFactory()`.

Niestety, obecnie nie są zdefiniowane żadne pliki odwzorowań. Jeśli chcesz, możesz skorzystać z przykładu „Witaj świecie" lub pominąć pozostałą część tego rozdziału i przejść do rozdziału 3. omawiającego klasy trwałości oraz odwzorowania. Czytaj dalej, jeśli chcesz poznać konfigurację Hibernate w środowisku zarządzanym.

2.3.3. Konfiguracja w środowisku zarządzanym

Środowisko zarządzane zajmuje się kilkoma najczęściej potrzebnymi zagadnieniami, na przykład bezpieczeństwem aplikacji (uwierzytelnianie i autoryzacja), pulami połączeń i zarządzaniem transakcjami. Typowym środowiskiem zarządzanym jest serwer aplikacji J2EE. Choć serwery aplikacji najczęściej projektuje się do obsługi EJB, można skorzystać z wielu udostępnianych przez nie usługi, nawet jeśli nie stosuje się ziarenek encyjnych EJB.

Hibernate najczęściej stosuje się w połączeniu z EJB sterowanym sesyjnie lub przez komunikaty, co przedstawia rysunek 2.4. EJB wykorzystuje te same interfejsy Hibernate co serwlet, JSP lub samowystarczalna aplikacja: `Session`, `Transaction` i `Query`. Kod dotyczący Hibernate można bez problemów przenosić między środowiskami zarządzanymi i niezarządzanymi. Sam system w niewidoczny sposób zajmuje się różnymi strategiami połączeń i transakcji.



Rysunek 2.4. Hibernate w środowisku zarządzanym przez serwer aplikacji

Serwer aplikacji udostępnia pulę połączeń jako **źródło danych** dowiązywane dzięki JNDI — egzemplarz `javax.jdbc.Datasource`. Należy poinformować Hibernate, gdzie ma szukać źródła danych w JNDI, podając pełną nazwę JNDI. Listing 2.5 przedstawia przykład pliku konfiguracyjnego dla wspomnianej sytuacji.

Listing 2.5. Przykładowy plik hibernate.properties dla źródła danych zapewnianego przez kontener

```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class =
net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
  net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
```

Plik najpierw podaje nazwę JNDI źródła danych. Źródło danych należy skonfigurować w deskryptorze wdrożenia aplikacji J2EE — sposób ustawiania zależy od dostawcy serwera aplikacji. Następnie włączona zostaje integracja Hibernate

z JTA. W następnym wierszu wskazujemy Hibernate lokalizację klasy `TransactionManager` serwera aplikacji, by zapewnić pełną integrację z transakcjami kontenera. Standard J2EE nie definiuje jednego konkretnego podejścia, ale Hibernate zawiera wbudowaną obsługę wszystkich popularnych serwerów aplikacji. Na końcu ponownie pojawia się informacja o dialekcie języka SQL.

Po poprawnej konfiguracji wszystkich elementów sposób korzystania z Hibernate w środowisku zarządzanym nie różni się znacząco od korzystania z wersji niezarządzanej — tworzy się obiekt `Configuration` z odwzorowaniami i na jego podstawie tworzy obiekt `SessionFactory`. Dodatkowej uwagi wymagają ustawienia związane z transakcyjnością środowiska.

Java ma standardowy interfejs programistyczny dla transakcji, JTA, który służy do sterowania transakcjami w zarządzanym środowisku J2EE. Są to tak zwane **transakcje zarządzane przez kontener** (CMT — *Container Managed Transaction*). Jeśli menedżer transakcji JTA występuje, połączenia JDBC są z nim związane i są pod jego pełną kontrolą. W środowisku niezarządzanym sytuacja jest zupełnie inna, ponieważ aplikacja (lub pula) bezpośrednio zarządza połączeniami i transakcjami JDBC.

Zarządzane i niezarządzane środowiska stosują inne podejście do transakcji. Ponieważ Hibernate w miarę możliwości chce zapewnić przenośność między oboma środowiskami, sam definiuje interfejs programistyczny do sterowania transakcjami. Interfejs `Transaction` z Hibernate stanowi abstrakcję ukrywającą różnice między transakcjami JTA i JDBC (a nawet transakcjami CORBA). Konkretną strategię transakcyjną ustawia właściwość `hibernate.connection.factory_ class`. Przyjmuje ona jedną z dwóch wartości:

- `net.sf.hibernate.transaction.JDBCTransactionFactory`, która deleguje wykonanie zadań do transakcji JDBC; strategię tę stosować należy z pulami połączeń w środowisku niezarządzanym (staje się domyślna, gdy nie określi się żadnej strategii).
- `net.sf.hibernate.transaction.JTATransactionFactory`, która deleguje wykonanie zadań do transakcji JTA; jest to poprawna strategia dla CMT, gdzie połączenia są powiązane z JTA. Warto pamiętać, że jeśli właśnie trwa transakcja JTA w momencie wywołania `beginTransaction()`, kolejne zadania zostają wykonane w tej transakcji (w przeciwnym razie powstaje nowa transakcja JTA).

Bardziej szczegółowe wprowadzenie do interfejsu programistycznego `Transaction` i wpływu wybranych scenariuszy transakcji zostało opisane w podrozdziale 5.1. Należy pamiętać o dwóch krokach koniecznych do przeprowadzenia w serwerze aplikacji J2EE: ustawieniu klasy fabryki dla interfejsu programistycznego `Transaction` na JTA w opisany wcześniej sposób i zadeklarowaniu wyszukiwania menedżera transakcji w sposób specyficzny dla serwera aplikacji. Strategia wyszukiwania potrzebna jest tylko wtedy, gdy korzysta się w Hibernate z buforowania dwupoziomowego. Nie zaszkodzi ustawiać jej także wtedy, gdy nie używa się takiego buforowania.

Hibernte z serwerem Tomcat

Tomcat nie jest pełnym serwerem aplikacji, ale kontenerem serwletów. Z drugiej strony zawiera pewne funkcje spotykane tylko w serwerach aplikacji. Hibernate może skorzystać z jednej z tych dodatkowych funkcji — puli połączeń Tomcata. Tomcat wewnętrznie korzysta z puli połączeń DBCP, ale udostępnia ją jako źródło danych JNDI (podobnie jak serwery aplikacji). Aby skonfigurować źródło danych Tomcata, dokonaj edycji pliku server.xml zgodnie z instrukcjami podanymi w dokumentacji JNDI/JDBC Tomcata. Hibernate odnajdzie źródło danych po ustawieniu właściwości hibernate.connection.datasource. Warto pamiętać, że Tomcat nie zawiera menedżera transakcji, więc cała sytuacja jest nadaj bardziej podobna do środowiska niezarządzanego.

W tej chwili powinieneś mieć działający system Hibernate, niezależnie od tego, czy używasz prostego kontenera serwletów czy serwera aplikacji. Utwórz i skompiluj klasę trwałości (na przykład klasę `Message` z początku rozdziału), skopiuj Hibernate i wymagane przez niego biblioteki do ścieżki wyszukiwania klas aplikacji wraz z plikiem `hibernate.properties`. Na końcu utwórz obiekt `SessionFactory`.

Kolejny podrozdział opisuje zaawansowane opcje konfiguracyjne Hibernate. Niektóre z nich polecamy. W szczególności możliwość tworzenia dla celów testowych dziennika wykonanych poleceń SQL lub stosowania wygodnych plików konfiguracyjnych XML zamiast prostych plików tekstowych. Można bezpiecznie pominąć kolejny podrozdział i powrócić do niego dopiero po zdobyciu w rozdziale 3. szczegółowej wiedzy na temat klas trwałych.

2.4. Zaawansowane ustawienia konfiguracyjne

Gdy aplikacja z Hibernate działa poprawnie, warto zainteresować się wszystkimi parametrami konfiguracyjnymi Hibernate. Dzięki nim nierzadko udaje się zoptymalizować działanie Hibernate, w szczególności przez odpowiednie dostrojenie współpracy z JDBC (na przykład zastosowanie aktualizacji wsadowych JDBC).

Nie chcemy na razie przynudzać wszystkimi szczegółami — najlepszym źródłem informacji na temat wszystkich opcji konfiguracyjnych jest dokumentacja Hibernate. W poprzednim podrozdziale przedstawiliśmy jedynie opcje pozwalające uruchomić aplikację.

Istnieje pewien parametr, na który już w tym momencie **musimy** kłaść duży nacisk. Przydaje się wielokrotnie, gdy tworzy się oprogramowanie. Ustawienie właściwości `hibernate.show_sql` na wartość `true` włącza wyświetlanie wszystkich utworzonych poleceń SQL na konsoli. Warto korzystać z tej opcji w momencie poszukiwania błędów, poszukiwania wąskiego gardła lub po prostu analizy działania Hibernate. Warto wiedzieć, czym tak naprawdę zajmuje się warstwa ORM, więc system nie ukrywa szczegółów poleceń SQL przed programistą.

Do tej pory zakładaliśmy przekazywanie parametrów konfiguracyjnych dzięki plikowi *hibernate.properties* lub programowo dzięki egzemplarzowi `java.util.Properties`. Trzecie z rozwiązań polega na użyciu pliku konfiguracyjnego w formacie XML.

2.4.1. Konfiguracja bazująca na pliku XML

Warto użyć pliku konfiguracyjnego XML (patrz listing 2.6), by w pełni skonfigurować obiekt `SessionFactory`. W odróżnieniu od pliku *hibernate.properties*, który zawiera tylko parametry konfiguracyjne, plik *hibernate.cfg.xml* może również określać lokalizacje dokumentów odwzorowań. Wiele osób preferuje centralizację konfiguracji Hibernate właśnie w ten sposób, by uniknąć dodawania parametrów do obiektu `Configuration` w kodzie aplikacji.

Listing 2.6. Przykładowy plik konfiguracyjny hibernate.cfg.xml

```
?xml version='1.0'encoding='utf-8'?>
<!DOCTYPE hibernate-configuration
PUBLIC "-//Hibernate/Hibernate Configuration DTD//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-2.0.dtd">
<hibernate-configuration>
  <session-factory name="java:/hibernate/HibernateFactory">
    <property name="show_sql">true</property>
    <property name="connection.datasource">
      java:/comp/env/jdbc/AuctionDB
    </property>
    <property name="dialect">
      net.sf.hibernate.dialect.PostgreSQLDialect
    </property>
    <property name="transaction.manager_lookup_class">
      net.sf.hibernate.transaction.JBossTransactionManagerLookup
    </property>
    <mapping resource="auction/Item.hbm.xml"/>
    <mapping resource="auction/Category.hbm.xml"/>
    <mapping resource="auction/Bid.hbm.xml"/>
<  /session-factory>
</hibernate-configuration>
```

Deklaracja typu dokumentu ❶

Atrybut ❷ nazwy

Specyfikacja ❸ właściwości

❹ Specyfikacja dokumentu odwzorowania

❶ Deklaracja **typu dokumentu** pomaga analizatorowi XML dokonać walidacji dokumentu, czyli sprawdzić zgodność jego formatu z wytycznymi zawartymi w DTD pliku konfiguracyjnego Hibernate.

❷ Opcjonalny atrybut `name` jest równoważny właściwości `hibernate.ses-sion_factory_name`. Służy do dowiązania JNDI dla `SessionFactory`, co zostanie omówione w dalszej części rozdziału.

❸ Właściwości hibernate można określać bez przedrostka `hibernate`. Poza tym przedrostkiem nazwy i wartości są dokładnie takie same jak we właściwościach ustawianych programowo.

❹ Dokumenty odwzorowań określa się jako zasoby aplikacji lub nawet jako na stałe zakodowane nazwy plików. Pliki użyte w przykładzie pochodzą z aplikacji systemu akcyjnego opisywanego w rozdziale 3.

Do inicjalizacji Hibernate wystarczy teraz użycie następującego kodu:

```
SessionFactory sessions = new Configuration()
    .configure().buildSessionFactory();
```

Chwileczkę — skąd Hibernate wie, gdzie znajduje się plik konfiguracyjny?

W momencie wywołania metody `configure()` Hibernate poszukuje pliku o nazwie *hibernate.cfg.xml* w ścieżce wyszukiwania klas. Jeśli chce się dla pliku konfiguracyjnego zastosować inną nazwę lub też plik znajduje się w niestandardowej lokalizacji, do metody `configure()` należy przekazać ścieżkę.

```
SessionFactory sessions = new Configuration()
    .configure("/hibernate-config/auction.cfg.xml")
    .buildSessionFactory();
```

Stosowanie plików konfiguracyjnych okazuje się bardziej wygodne od plików właściwości lub programowego ustawiania opcji. Możliwość usunięcia z kodu lokalizacji plików odwzorowań klas (nawet jeśli miałby się znaleźć w pomocniczej klasie inicjalizującej) stanowi ogromną zaletę przedstawionego podejścia. Nic nie stoi na przeszkodzie, by stosować różne zbiory plików odwzorowań (i inne opcje konfiguracyjne) w zależności od bazy danych i środowiska (produkcyjnego lub testowego). Co więcej, można je przełączać programowo.

Jeśli ścieżka wyszukiwania klas zawiera pliki *hibernate.properties* i *hibernate.cfg.xml*, ustawienia występujące w pliku XML nadpiszą ustawienia z pliku właściwości. Rozwiązanie bywa użyteczne, gdy pewne podstawowe ustawienia trzyma się w pliku właściwości i modyfikuje dla każdego wdrożenia plikiem XML.

Warto zwrócić uwagę na nadanie parametru `name` dla obiektu `SessionFactory` w pliku konfiguracyjnym XML. Hibernate używa tej nazwy do automatycznego dowiązania obiektu do JNDI po jego utworzeniu.

2.4.2. Obiekt SessionFactory dowiązany do JNDI

W większości aplikacji stosujących Hibernate obiekt `SessionFactory` powinien zostać utworzony tylko jeden raz w momencie inicjalizacji aplikacji. Cała aplikacja powinna stosować ten pojedynczy egzemplarz — wszystkie obiekty Session należy tworzyć na jego podstawie. Często pojawia się pytanie, gdzie umieścić obiekt fabryki, by był bez trudu dostępny z każdego miejsca aplikacji.

W środowisku J2EE obiekt `SessionFactory` warto dowiązać do JNDI, by łatwo udostępnić go wielu wątkom i komponentom wykorzystującym Hibernate. Oczywiście JNDI to nie jedyny sposób, w jaki komponenty aplikacji mogą uzyskać obiekt `SessionFactory`. Istnieje wiele możliwych implementacji wzorca rejestru, włączając w to użycie obiektu `ServletContext` lub zmiennej `static final` w obiekcie singletonu. Szczególnie eleganckim wydaje się być rozwiązanie stosujące komponent szkieletowy odwrócenia sterowania (IoC — *Inversion of Control*) o zasięgu całej aplikacji. JNDI jest popularnym rozwiązaniem udostępnianym jako usługi JMX, co wkrótce pokażemy. Niektóre alternatywy przedstawimy w podrozdziale 8.1.

Obiekt `SessionFactory` automatycznie dowiąże się do JNDI, gdy właściwość `hibernate.session_factory_name` będzie zawierała nazwę węzła katalogu. Jeśli środowisko wykonawcze nie zapewnia domyślnego kontekstu JNDI (lub też domyślna implementacja JNDI nie obsługuje egzemplarzy interfejsu `Reference-`

able), należy wskazać kontekst JNDI, używając właściwości `hibernate.jndi.url` i `hibernate.jndi.class`.

Uwaga Interfejs programistyczny JNDI umożliwia zapisywanie i odczyt obiektów w strukturze hierarchicznej (drzewiastej). JNDI implementuje wzorzec rejestru. Z JNDI można powiązać obiekty infrastruktury (konteksty transakcji, źródła danych), ustawienia konfiguracyjne (opcje środowiska, rejestry użytkownika) lub nawet obiekty aplikacji (referencje do EJB, fabryki obiektów).

Poniżej przedstawiamy przykładową konfigurację Hibernate, która dowiązuje obiekt `SessionFactory` do nazwy `hibernate/HibernateFactory`, wykorzystując bezpłatną implementację JNDI firmy Sun (*fscontext.jar*) bazującą na systemie plików.

```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class =
net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class = \
net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
hibernate.session_factory_name = hibernate/HibernateFactory
hibernate.jndi.class = com.sun.jndi.fscontext.RefFSContextFactory
hibernate.jndi.url = file:/auction/jndi
```

Do określenia parametrów równie dobrze można skorzystać z pliku konfiguracyjnego XML. Przykład jest mało realistyczny, bo większość serwerów aplikacji udostępniających pule połączeń dzięki JNDI stosuje również implementację JNDI z domyślnym kontekstem dopuszczającym zapis. Opisane podejście stosuje serwer JBoss, w którym można opuścić dwie ostatnie właściwości i określić tylko nazwę dla obiektu `SessionFactory`. Do inicjalizacji dowiązania wystarczy wtedy wykonanie kodu `Configuration().configure().buildSessionFactory()`.

Uwaga Tomcat zawiera wbudowany kontekst JNDI tylko do odczytu — nie można w nim nic zapisać z poziomu aplikacji po uruchomieniu kontenera serwletów. Z tego powodu Hibernate nie potrafi skorzystać z tego kontekstu — należy stosować pełną implementację kontekstu (wykorzystując wspomniany kontekst firmy Sun) lub wyłączyć dowiązanie JNDI dla obiektu `SessionFactory` przez pominięcie w konfiguracji właściwości `session_factory_name`.

Przyjrzyjmy się innym istotnym opcjom konfiguracyjnym Hibernate dotyczącym tworzenia dziennika operacji.

2.4.3. Dzienniki

Hibernate (i wiele innych implementacji ORM) wykonuje polecenia SQL w sposób **asynchroniczny**. Polecenie `INSERT` zazwyczaj nie jest wykonywane zaraz po wywołaniu przez aplikację metody `Session.save()`. Podobnie polecenie `UPDATE`

nie wykona się od razu po wywołaniu `Item.addBid()`. Instrukcje SQL zostają wykonane na końcu transakcji. Takie zachowanie nazywa się **zapisem opóźnionym**.

Takie podejście znacząco utrudnia śledzenie i debugowanie kodu ORM. W teorii możliwe jest traktowanie przez aplikację systemu Hibernate jako czarnej skrzynki i ignorowanie jego zachowania. Aplikacja nie potrafi wykryć asynchroniczności działań Hibernate (a przynajmniej nie bez posiłkowania się bezpośrednimi wywołaniami JDBC). Gdy pojawiają się problemy, warto byłoby się dowiedzieć, co tak **naprawdę** wykonuje Hibernate i w jakiej kolejności. Ponieważ Hibernate jest projektem typu *open source*, można przyjrzeć się jego kodowi źródłowemu. Czasem to naprawdę pomaga! Z drugiej strony z powodu asynchronicznego działania Hibernate przy jego debugowaniu łatwo się „stracić". Zamiast korzystać z debugera, lepiej użyć dziennika operacji.

Wspomnieliśmy wcześniej o parametrze konfiguracyjnym `hibernate.show_sql`, który warto włączyć za każdym razem, gdy pojawią się problemy. Czasem polecenia SQL nie wystarczają i trzeba sięgnąć głębiej.

Hibernate tworzy dziennik wszystkich interesujących operacji, używając biblioteki `commons-logging` Apache. Jest to cienka warstwa abstrakcji kierująca dane do biblioteki log4j Apache (jeśli plik *log4j.jar* znajduje się w ścieżce wyszukiwania klas) lub do systemu dziennika z JDK 1.4 (jeśli używa się Javy 1.4 lub nowsze i brakuje log4j). Zalecamy stosowanie log4j, gdyż jest bardziej rozwinięty, bardziej popularny i stale rozwijany.

Aby zobaczyć jakiekolwiek wyniki działania log4j, w ścieżce wyszukiwania klas potrzeba pliku *log4j.properties* (podobnie jak plików *hibernate.properties* i *hibernate.cfg.xml*). Poniższa przykładowa zawartość tego pliku powoduje wyświetlanie wszystkich komunikatów na konsoli.

```
### kieruj komunikaty bezpośrednio na standardowe wyjście ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.Target=System.out
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE}%5p
      %c{1}:%L - %m%n
### opcje dziennika podstawowego ###
log4j.rootLogger=warn, stdout
### opcje dziennika Hibernate ###
log4j.logger.net.sf.hibernate=info
### uwzględniaj parametry dowiązań JDBC ###
log4j.logger.net.sf.hibernate.type=info
### uwzględniaj aktywność bufora poleceń przygotowanych ###
log4j.logger.net.sf.hibernate.ps.PreparedStatementCache=info
```

Przy tej konfiguracji w trakcie działania systemu nie pojawi się zbyt wiele komunikatów. Wystarczy jednak zamienić wartość `info` na `debug` w kategorii `log4j.logger.net.sf.hibernate`, by poznać szczegóły działania Hibernate. Warto się upewnić, że tego rodzaju konfiguracja nie trafi na serwer produkcyjny — zapis do dziennika będzie wolniejszy od dostępu do bazy danych.

Konfiguracja zawiera się obecnie w trzech plikach: *hibernate.properties*, *hibernate.cfg.xml* i *log4j.properties*.

Istnieje jeszcze jeden sposób konfiguracji Hibernate, jeśli serwer aplikacji obsługuje JMX (*Java Management Extensions*).

2.4.4. Java Management Extensions

Świat Javy pełen jest specyfikacji, standardów i ich implementacji. Stosunkowo nowym i istotnym już od pierwszej wersji jest standard **Java Management Extensions** (JMX). Dotyczy on zarządzania komponentami systemowymi, a w zasadzie usługami systemu.

Gdzie w tym całym obrazie mieści się Hibernate? Gdy umieści się go na serwerze aplikacji, korzysta z innych usług, na przykład zarządzania transakcjami i pulą połączeń z bazą danych. Dlaczego nie uczynić samego Hibernate zarządzalną usługą, której mogą używać inne elementy systemu? Integracja z JMX czyni z Hibernate zarządzalny komponent JMX.

Specyfikacja JMX definiuje następujące komponenty:

- **JMX MBean** — komponent wielokrotnego użytku (najczęściej infrastrukturalny), który udostępnia interfejs zarządzania (administracji),
- **kontener JMX** — zapewnia uogólniony dostęp (lokalny lub zdalny) do MBean,
- **klient JMX** (najczęściej uogólniony) — umożliwia administrację dowolnym MBean dzięki kontenerowi JMX.

Serwer aplikacji obsługujący JMX (na przykład JBoss) działa jak kontener JMX, umożliwiając konfigurację i inicjalizację komponentów MBean jako części procesu uruchamiania serwera aplikacji. Można monitorować i administrować MBean, stosując konsolę administracyjną serwera aplikacji (działa jak klient JMX).

Komponent MBean daje się utworzyć jako usługę JMX, która nie tylko jest przenośnia między różnymi serwerami aplikacji z wbudowaną obsługą JMX, ale również zapewnia wdrożenie w działającym systemie (tak zwane „wdrożenie na gorąca").

Hibernate można administrować jako JMX MBean. Usługa JMX dopuszcza inicjalizację Hibernate w momencie uruchamiania serwera aplikacji i sterowania (konfigurowania) go dzięki klientowi JMX. Z drugiej strony komponenty JMX nie są automatycznie zintegrowane z transakcjami zarządzanymi przez kontener. Z tego powodu opcje konfiguracyjne z listingu 2.7 (deskryptor wdrożenia usługi w JBoss) wyglądają podobnie do ustawień Hibernate dla środowiska zarządzanego.

Listing 2.7. Deskryptor wdrożenia Hibernate jako JMX w JBoss — plik jboss-service.xml

```
<server>
<mbean code="net.sf.hibernate.jmx.HibernateService"
    name="jboss.jca:service=HibernateFactory, name=HibernateFactory">
  <depends>jboss.jca:service=RARDeployer</depends>
  <depends>jboss.jca:service=LocalTxCM,name=DataSource</depends>
  <attribute name="MapResources">
    auction/Item.hbm.xml, auction/Bid.hbm.xml
```

```
    </attribute>
    <attribute name="JndiName">
      java:/hibernate/HibernateFactory
    </attribute>
    <attribute name="Datasource">
      java:/comp/env/jdbc/AuctionDB
    </attribute>
    <attribute name="Dialect">
      net.sf.hibernate.dialect.PostgreSQLDialect
    </attribute>
    <attribute name="TransactionStrategy">
      net.sf.hibernate.transaction.JTATransactionFactory
    </attribute>
    <attribute name="TransactionManagerLookupStrategy">
      net.sf.hibernate.transaction.JBossTransactionManagerLookup
    </attribute>
    <attribute name="UserTransactionName">
      java:/UserTransaction
    </attribute>
  </mbean>
  </server>
```

Usługa `HibernateService` zależy od dwóch innych usług JMX — `service=RARDeployer` i `service=LocalTxCM,name=DataSource` — znajdujących się w domenie usługowej `jboss.jca`.

MBean dla Hibernate znajduje się w pakiecie `net.sf.hibernate.jmx`. Niestety, metody zarządzania cyklem życia JMX nie stanowią części specyfikacji JMX 1.0. Z tego powodu metody `start()` i `stop()` z `HibernateService` są specyficzne dla serwera aplikacji JBoss.

Uwaga Jeśli jesteś zainteresowany zaawansowanymi użyciami JMX, warto zajrzeć do kodu serwera JBoss. Wszystkie usługi (nawet kontener EJB) z JBoss są zaimplementowane jako MBean i mogą być zarządzane dzięki interfejsowi konsolowemu.

Zalecamy konfigurację Hibernate w sposób programowy (z zastosowaniem obiektu `Configuration`) przez próbą jego uruchomienia jako usługa JMX. Niektóre funkcje (na przykład wdrażanie na gorąco aplikacji z Hibernate) jest możliwe tylko z JMX. Największą zaletą Hibernate działającego z JMX jest automatyczny rozruch. Nie trzeba ręcznie tworzyć obiektów `Configuration` i SessionFactory. Wystarczy korzystać z obiektu `SessionFactory` przez JNDI zaraz po uruchomieniu `HibernateService`.

2.5. Podsumowanie

W rozdziale przyjrzeliśmy się Hibernate z nieco wyższego poziomu. Dodatkowo przedstawiliśmy jego architekturę, stosując przykład „Witaj świecie". Pokazaliśmy, jak konfigurować Hibernate w różnych środowiskach z zastosowaniem rozmaitych sposobów przekazywania konfiguracji, włączając w to JMX.

Interfejsy `Configuration` i `SessionFactory` to punkty początkowe dla aplikacji chcących używać Hibernate zarówno w środowisku zarządzanym, jak i niezarządzanym. Hibernate stosuje dodatkowe interfejsy programistyczne, na przykład interfejs `Transaction`, by zakryć różnice środowisk i zapewnić przenośność kodu trwałości danych.

Hibernate można zintegrować z niemal dowolnym środowiskiem Javy: serwletem, apletem lub trójwarstwową aplikacją klient-serwer. Najważniejszymi elementami konfiguracji Hibernate są: zasoby bazy danych (konfiguracja połączeń), strategie transakcyjne i oczywiście metadane odwzorowań zawarte w plikach XML.

Interfejsy konfiguracyjne Hibernate zostały tak zaprojektowane, by obsłużyć możliwe dużo różnych sposobów użycia przy zachowaniu ich dużej zrozumiałości. Najczęściej jeden plik o nazwie *hibernate.cfg.xml* i jeden wiersz kodu w programie wystarczają do uruchomienia Hibernate.

Konfiguracja nie miałaby jednak większego sensu bez klas trwałych i dokumentów odwzorowań. Kolejny rozdział został w całości poświęcony pisaniu i odwzorowywaniu klas trwałych. Wkrótce się przekonasz, jak w rzeczywistych aplikacjach dzięki nietrywialnemu odwzorowaniu obiektowo-relacyjnemu zapewnić trwałość obiektów.