Hibernate

Od Nowicjusza do Profesjonalisty

Dave Minter
Jeff Linwood

POWERnet

Apress®

Beginning Hibernate: From Novice to Professional

ISBN-13 (pbk): 978-1-59059-693-7
ISBN-10 (pbk): 1-59059-693-5

Hibernate. Od Nowicjusza do Profesjonalisty

ISBN: 978-83-924603-0-5

Kod źródłowy dla tej książki jest dostępny dla czytelników na stronie `http://www.apress.com` w dziale „Source Code".

ROZDZIAŁ 3

Tworzenie prostej aplikacji

W tym rozdziale przyjrzymy się jeszcze raz niektórym czynnościom potrzebnym, aby wprowadzić w życie przykład z pierwszego rozdziału. Stworzymy także od podstaw nieco większą aplikację. Wszystkie kody zamieszczone w tej książce można pobrać z witryny wydawnictwa Apress (`www.apress.com`).

Instalowanie narzędzi

Aby móc pracować na przykładach podanych w tym rozdziale, konieczna jest instalacja kilku programów narzędziowych. Potrzebne będą: JDK, Hibernate i Hibernate Tools, a także narzędzie kompilacji Ant i baza danych HSQLDB. Tabela 3.1 zawiera listę potrzebnych narzędzi oraz adresy stron, na których można je znaleźć.

Tabela 3.1. *Programy narzędziowe wykorzystane w tej książce*

Narzędzie	Wersja	Adres strony
Hibernate	3.2.0	`http://hibernate.org`
Hibernate Tools	3.1	`http://hibernate.org`
Ant	1.6.5	`http://ant.apache.org`
HSQLDB	1.8.0.2	`http://hsqldb.org`

Hibernate i Hibernate Tools

Najnowsza wersja Hibernate'a jest zawsze dostępna na stronie `http://hibernate.org` (odsyłacz "Download" znajdujący się w menu po lewej stronie). Pod tym odnośnikiem dostępne są liczne starsze wersje i dodatkowe biblioteki, lecz należy wybrać Hibernate Core 3.2.0 lub wersję późniejszą. W czasie pisania niniejszej książki wersja ta nie została jeszcze udostępniona, ale można się spodziewać, że nastąpi to zanim książka trafi na półki w księgarni. Jeśli tak się jednak nie stanie, większość przykładów będzie również dobrze działała z wersją 3.1.0. Należy pobrać plik zarchiwizowany i rozpakować go do lokalnego katalogu. Ten skompresowany plik zawiera wszystkie kody źródłowe samego Hibernate'a, a także bibliotekę JAR stworzoną z tego źródła i wszystkie pliki biblioteki potrzebne do uruchomienia przykładu.

Powinniśmy następnie pobrać Hibernate Tools z tej samej strony. W czasie pisania tej książki narzędzia te dostępne są w wersji 3.1 (jest to wydanie beta, ale jeśli ostateczna wersja 3.1 nie będzie jeszcze udostępniona, zaleca się korzystanie z niego zamiast z jego gorszych

poprzedników). Hibernate Tools zawierają rozmaite wtyczki (ang. *plug-ins*) dla Anta oraz darmowego środowiska Eclipse IDE. W tym rozdziale wykorzystujemy jedynie wtyczki Anta, natomiast cechy Eclipse omówimy w Dodatku B. I tym razem plik zarchiwizowany należy pobrać i rozpakować w lokalnym katalogu. Ten plik nie zawiera kodu źródłowego (który, w razie takiej potrzeby, jest udostępniony na stronie `www.hibernate.org`).

HSQLDB 1.8.0

W naszych przykładach będziemy wykorzystywali bazę danych HSQL. Jest ona napisana w Javie i stanowi darmowo dostępne oprogramowanie typu open source. W przykładach wykorzystaliśmy wersję 1.8.0.2, więc można oczekiwać, że wszystkie późniejsze wersje też będą odpowiednie. HSQL czerpie z kodu pierwotnie wydanego jako "Hypersonic". Tę nazwę można niekiedy spotkać w dokumentacji HSQL i jest ona używana synonimicznie z określeniem "HSQL". Ta baza danych określana bywa również jako HSQLDB, aby nie pomylić jej z akronimem HQL (Hibernate Query Language), który jest łudząco podobny!

Nasze przykłady są dostosowane do bazy HSQL, ponieważ HSQL działa na każdej platformie, na której można uruchomić Hibernate'a, a także dlatego, iż HSQL jest bazą darmowo dostępną i mającą małe wymagania instalacyjne. Natomiast podczas uruchamiania przykładów z naszą własną bazą danych, różnice powinny sprowadzać się do następujących elementów:

- Klasa dialektowa Hibernate'a
- Sterownik JDBC
- Połączenie URL z bazą danych
- Nazwa użytkownika bazy danych
- Hasło do bazy danych

W dalszej części rozdziału powiemy, w jaki sposób możemy definiować te elementy. Zauważmy, że w miejscu, gdzie podajemy adres URL dla połączenia z bazą danych, często dopisujemy atrybut `shutdown=true`. Zapobiega to niewielkiemu problemowi, kiedy HSQLDB nie zapisuje zmian na dysku, jeśli obiekt Connection nie jest zamknięty. Sytuacja ta może nigdy nie wystąpić, jeżeli połączenie jest zarządzane przez samego Hibernate'a (jego własną logikę puli połączeń). Nie jest to konieczne w przypadku niewbudowanych baz danych.

Ant 1.6.5

Zainstalujmy narzędzie kompilacji Ant. Nie będziemy szczegółowo wyjaśniać formatu `build.xml`. Osobom znającym Anta wystarczy na początek przykładowy skrypt zamieszczony w tym rozdziale, natomiast dla pozostałych sam Ant jest już wystarczającym wyzwaniem. Godną polecenia książką, będącą dobrym opracowaniem narzędzi typu open source takich jak Ant, jest *Enterprise Java Development on a Budget*, której autorami są Christopher M. Judd i Brian Sam Bodden (wydawnictwo Apress, 2004).

Ponieważ tematyka dotycząca Anta wykracza poza zakres niniejszej książki, omówimy sposób wykorzystania zadań Hibernate'a użytych w naszych skryptach.

Listing 3.1 podaje skrypt narzędzia Ant do skompilowania przykładu z tego rozdziału.

Listing 3.1. *Skrypt Anta do skompilowania przykładów z rozdziału trzeciego*

```
<project name="sample">

    <property file="build.properties"/>

    <property name="src" location="src"/>
    <property name="bin" location="bin"/>
    <property name="sql" location="sql"/>
    <property name="hibernate.tools"
        value="${hibernate.tools.home}${hibernate.tools.path}"/>

    <path id="classpath.base">
        <pathelement  location="${src}"/>
        <pathelement  location="${bin}"/>
        <pathelement  location="${hibernate.home}/hibernate3.jar"/>
        <fileset  dir="${hibernate.home}/lib"  includes="**/*.jar"/>
        <pathelement location="${hsql.home}/lib/hsqldb.jar"/>
    </path>

    <path id="classpath.tools">
        <path refid="classpath.base"/>
        <pathelement
            location="${hibernate.tools}/hibernate-tools.jar"/>
    </path>

    <taskdef name="htools"
        classname="org.hibernate.tool.ant.HibernateToolTask"
        classpathref="classpath.tools"/>

    <target name="exportDDL" depends="compile">
        <htools destdir="${sql}">
            <classpath refid="classpath.tools"/>
            <configuration
                configurationfile="${src}/hibernate.cfg.xml"/>
            <hbm2ddl drop="true" outputfilename="sample.sql"/>
        </htools>
    </target>

    <target name="compile">
        <javac  srcdir="${src}"  destdir="${bin}" classpathref="classpath.base"/>
    </target>
```

```
<target name="populateMessages" depends="compile">
    <java classname="sample.PopulateMessages" classpathref="classpath.base"/>
</target>

<target name="listMessages" depends="compile">
    <java  classname="sample.ListMessages"  classpathref="classpath.base"/>
</target>

<target name="createUsers" depends="compile">
    <java  classname="sample.CreateUser"  classpathref="classpath.base">
        <arg value="dave"/>
        <arg value="dodgy"/>
    </java>
    <java  classname="sample.CreateUser"  classpathref="classpath.base">
        <arg value="jeff"/>
        <arg value="jammy"/>
    </java>
</target>

<target name="createCategories" depends="compile">
    <java  classname="sample.CreateCategory" classpathref="classpath.base">
        <arg value="retro"/>
    </java>
    <java  classname="sample.CreateCategory" classpathref="classpath.base">
        <arg  value="kitsch"/>
    </java>
</target>

<target name="postAdverts" depends="compile">
    <java  classname="sample.PostAdvert"  classpathref="classpath.base">
        <arg value="dave"/>
        <arg value="retro"/>
        <arg  value="Sinclair  Spectrum  for  sale"/>
        <arg  value="48k  original  box  and  packaging"/>
    </java>
    <java  classname="sample.PostAdvert"  classpathref="classpath.base">
        <arg value="dave"/>
        <arg  value="kitsch"/>
        <arg  value="Commemorative  Plates"/>
        <arg value="Kitten and puppies design"/>
    </java>
    <java  classname="sample.PostAdvert"  classpathref="classpath.base">
        <arg value="jeff"/>
        <arg value="retro"/>
        <arg  value="Atari  2600  wanted"/>
        <arg  value="Must  have  original  joysticks."/>
```

```
        </java>
        <java  classname="sample.PostAdvert"  classpathref="classpath.base">
            <arg value="jeff"/>
            <arg  value="kitsch"/>
            <arg  value="Inflatable  Sofa"/>
            <arg  value="Leopard  skin  pattern.  Nice."/>
        </java>
    </target>

    <target name="listAdverts" depends="compile">
        <java  classname="sample.ListAdverts"  classpathref="classpath.base"/>
    </target>

</project>
```

Zaimportowany w pierwszej linii plik właściwości udostępnia ścieżki do naszych zainstalowanych bibliotek. Powinniśmy odpowiednio dostosować go (jak pokazano w listingu 3.2). Kiedy rozpakujemy Hibernate'a 3.2.0, utworzy on katalog o nazwie `hibernate-3.2`, którego nazwę zmieniliśmy na ścieżkę pełnej wersji. Coś podobnego zrobiliśmy z katalogiem bazy danych HSQL.

Archiwum Hibernate Tools rozpakowuje się aktualnie do dwóch katalogów (`plugins` i `features`). Zawarliśmy je w stworzonym przez nas katalogu nadrzędnym. Ścieżka do odpowiedniego pliku JAR (`hibernate-tools.jar`) w nierozpakowanym katalogu jest uzależniona od konkretnej wersji Hibernate Tools. Dlatego też dodaliśmy właściwość `hibernate.tools.path`, aby wskazać tę ścieżkę naszemu skryptowi kompilacji.

Listing 3.2. *Plik* `build.properties` *konfigurujący skrypt Anta*

```
# Path to the hibernate install directory
hibernate.home=/hibernate/hibernate-3.2.0

# Path to the hibernate-tools install directory
hibernate.tools.home=/hibernate/hibernate-tools-3.1

# Path to hibernate-tools.jar relative to hibernate.tools.home
hibernate.tools.path=/plugins/org.hibernate.eclipse_3.1.0/lib/tools

# Path to the HSQL DB install directory
hsql.home=/hsqldb/hsqldb-1.8.0.2
```

Poza ustawieniami konfiguracyjnymi, jedyną osobliwością pliku `build.xml` jest konfiguracja i użycie zadania Anta związanego z frameworkiem Hibernate. Parametr `taskdef` (ukazany w listingu 3.3) udostępnia nam to zadanie, wykorzystując odpowiednie klasy z pliku `tools.jar`.

Listing 3.3. *Definiowanie zadań Anta dla Hibernate Tools*

```
<taskdef name="htools"
    classname="org.hibernate.tool.ant.HibernateToolTask"
    classpathref="classpath.tools"/>
```

To zadanie udostępnia kilka podzadań, ale w tym rozdziale wykorzystamy tylko podzadanie `hbm2ddl`. Odczytuje ono pliki odwzorowań i pliki konfiguracyjne, oraz generuje skrypty Data Definition Language (DDL – ang. *„język definicji danych"*). Tworzy w ten sposób odpowiedni schemat bazy danych do reprezentowania naszych encji.

Tabela 3.2 prezentuje podstawowe katalogi przyjmowane przez nasz skrypt kompilacji pokrewne katalogowi głównemu naszego projektu.

Tabela 3.2. *Katalogi projektu*

Katalog	Zawartość
`src`	Kod źródłowy i pliki konfiguracyjne (z wyłączeniem tych bezpośrednio związanych z kompilacją)
`bin`	Pliki skompilowanych klas
`sql`	Wygenerowane skrypty DDL

Katalog główny projektu zawiera plik skryptu oraz konfiguracji kompilacji. Będzie zawierał również pliki bazy danych wygenerowane przez HSQL podczas uruchomienia zadania `exportDDL`.

Zadania narzędzia Ant

Tabela 3.3 pokazuje zadania, jakie zawiera skrypt kompilacji Anta.

Tabela 3.3. *Zadania dostępne w przykładowym skrypcie Anta*

Zadanie	Akcja
`exportDDL`	Tworzy odpowiednie obiekty bazy danych. Generuje także skrypt, który – jeśli to konieczne – może być uruchomiony względem bazy danych HSQL, aby odtworzyć te obiekty.
`compile`	Kompiluje pliki klas. To zadanie jest uzależnione od wszystkich innych zadań z wyjątkiem `exportDDL` (które nie wymaga plików klas). Nie jest zatem konieczne bezpośrednie wywoływanie go.
`populateMessages`	Wypełnia bazę danych przykładowym komunikatem.
`listMessages`	Wypisuje wszystkie komunikaty przechowywane w bazie danych przez zadanie `populateMessages`.

Zadanie	Akcja
`createUsers`	Tworzy dwóch użytkowników w bazie danych dla przykładu Advert (ang. *„reklama”*).
`createCategories`	Tworzy dwie kategorie w bazie danych dla przykładu Advert.
`postAdverts`	Tworzy kilka reklam w bazie danych dla przykładu Advert.
`listAdverts`	Wymienia reklamy w bazie danych dla przykładu Advert.

Umożliwianie logowania

Zanim uruchomimy w tym rozdziale jakikolwiek przykład, będziemy musieli stworzyć w ścieżce klas plik `log4j.properties`. Odpowiedni przykład jest dostarczony wraz z narzędziami Hibernate’a w katalogu `etc` nierozpakowanego archiwum.

Nasz przykład włącza ten plik do katalogu `src` naszego projektu i ten właśnie katalog umieszcza w ścieżce klas. W pewnych okolicznościach – takich jak tworzenie pliku JAR do włączenia w innych projektach – lepszym rozwiązaniem może być skopiowanie odpowiedniego pliku (lub plików) właściwości do katalogu docelowego z plikami klas.

Tworzenie pliku konfiguracyjnego Hibernate’a

Hibernate w różny sposób może uzyskać wszystkie informacje potrzebne mu do połączenia z bazą danych i określenia jej odwzorowań. W naszym przykładzie Message wykorzystaliśmy plik konfiguracyjny `hibernate.cfg.xml` umieszczony w katalogu `src` naszego projektu i podany w listingu 3.4.

Listing 3.4. *Plik odwzorowań aplikacji Message*

```
<?xml version='1.0' encoding='utf-8'?>
    <session-factory>
        <property name="hibernate.connection.url">
            jdbc:hsqldb:file:testdb;shutdown=true
        </property>
        <property name="hibernate.connection.driver_class">
            org.hsqldb.jdbcDriver
        </property>
        <property name="hibernate.connection.username">sa</property>
        <property name="hibernate.connection.password"></property>
        <property name="hibernate.connection.pool_size">0</property>
        <property name="hibernate.dialect">
            org.hibernate.dialect.HSQLDialect
        </property>
        <property name="hibernate.show_sql">false</property>
```

```
        <!-- "Import" the mapping resources here -->
        <mapping resource="sample/entity/Message.hbm.xml"/>

    </session-factory>
</hibernate-configuration>
```

Różne pola związane z bazami danych (`hibernate.connection.*`) powinny być nam dosyć znane z ustawiania połączeń JDBC, z wyjątkiem właściwości `hibernate.connection.pool`. Wykorzystywany jest on do wyłączania funkcji puli połączeń, która powoduje problemy podczas używania bazy danych HSQL. Niezwykle użyteczna podczas diagnozowania problemów okazuje się wartość `show_sql` (w naszym przykładzie ustawiona na `false`). Jeśli jest ona ustawiona na `true`, cały SQL przygotowany przez Hibernate'a jest logowany do standardowego strumienia wyjścia (czyli do konsoli).

Dialekty SQL, przedstawione w drugim rozdziale, umożliwiają nam wybór typu bazy danych, z którą Hibernate będzie się komunikował. Musimy wybrać dialekt, nawet jeżeli będzie to `GenericDialect`. Większość platform baz danych przyjmuje popularny podzbiór SQL, jednakże każda z nich odznacza się charakterystycznymi dla niej niezgodnościami i rozwiązaniami niestandardowymi. Hibernate wykorzystuje klasę dialektów do określenia odpowiedniego dialektu SQL, aby użyć go podczas tworzenia bazy danych i wykonywania zapytań. Jeżeli zdecydujemy się na `GenericDialect`, wówczas Hibernate będzie mógł wykorzystać tylko wspólny podzbiór SQL do wykonywania operacji. Nie będzie on mógł wykorzystać charakterystycznych cech rozmaitych baz danych zwiększających wydajność.

■ **Uwaga** Hibernate wyszukuje plik konfiguracyjny w ścieżce klas (classpath). Jeżeli zamieścimy go w jakimkolwiek innym miejscu, Hibernate zaprotestuje ze względu na brak koniecznych szczegółów konfiguracyjnych.

Hibernate nie wymaga użycia pliku konfiguracyjnego XML. Dysponujemy dwoma opcjonalnymi rozwiązaniami. Po pierwsze, możemy udostępnić zwykły plik właściwości Javy. Odpowiedni plik właściwości do listingu 3.4 będzie wyglądał następująco:

```
hibernate.connection.driver_class=org.hsqldb.jdbcDriver
hibernate.connection.url=jdbc:hsqldb:file:testdb;shutdown=true
hibernate.connection.username=sa
hibernate.connection.password=
hibernate.connection.pool_size=0
hibernate.show_sql=false
hibernate.dialect=org.hibernate.dialect.HSQLDialect
```

Jak widzimy, nie zawiera on odwzorowań zasobów z pliku XML – i tak naprawdę nie możemy zamieścić tej informacji w pliku właściwości. Jeżeli w ten sposób chcemy skonfigurować Hibernate'a, będziemy musieli bezpośrednio odwzorować nasze klasy do obiektu `Configuration` w czasie wykonania. Oto, w jaki sposób możemy to zrobić:

```
Configuration config = new Configuration();
config.addClass( sample.entity.Message.class );
config.setProperties( System.getProperties() );
SessionFactory sessions = config.buildSessionFactory();
```

Zauważmy, że obiekt `Configuration` będzie poszukiwał w ścieżce klas pliku odwzorowań z tego samego pakietu co klasa, która została przekazana. W tym przykładzie zatem, gdzie posiadającą pełny kwalifikator nazwą klasy jest `sample.entity.Message`, powinna znajdować się poniższa para plików z katalogu głównego ścieżki klas:

```
/sample/entity/Message.class
/sample/entity/Message.hbm.xml
```

Klasa `Message.class` stanowi skompilowane dane wyjściowe z kodu `Message.java` podanego w listingu 3.5 (i pokrótce omówionego w rozdziale pierwszym). `Message.hbm.xml` jest z kolei plikiem odwzorowań XML ukazanym w listingu 1.5 z rozdziału pierwszego. Jeżeli z jakiegoś powodu chcemy trzymać nasze pliki odwzorowań w innym katalogu, możemy także w następujący sposób udostępnić je jako zasoby (zauważmy, że ścieżka tego zasobu musi nadal być powiązana ze ścieżką klas):

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

Możemy mieć tak wiele (lub tak niewiele) plików odwzorowań, jak sobie tego życzymy, nadając im nazwy według naszych upodobań. Przyjęło się jednakże posiadanie jednego pliku odwzorowań dla każdej odwzorowanej klasy, umieszczonego razem z klasą w tym samym katalogu i nazwanego podobnie do niej (np. plik `Message.hbm.xml` z domyślnego pakietu odwzorowujący klasę `Message` także w domyślnym pakiecie). Pozwala to na szybkie odnalezienie odwzorowania jakiejkolwiek klasy oraz na łatwy odczyt plików odwzorowań.

Jeżeli nie chcemy udostępniać właściwości konfiguracyjnych w pliku, możemy bezpośrednio ustawić je wykorzystując flagę `-D`. Oto przykład:

```
java -classpath ...
    -Dhibernate.connection.driver_class=org.hsqldb.jdbcDriver
    -Dhibernate.connection.url= jdbc:hsqldb:file:testdb;shutdown=true
    -Dhibernate.connection.username=sa
    -Dhibernate.connection.password=
    -Dhibernate.connection.pool_size=0
    -Dhibernate.show_sql=false
    -Dhibernate.dialect=org.hibernate.dialect.HSQLDialect
     ...
```

Biorąc pod uwagę rozmiar kodu, jest to prawdopodobnie najmniej praktyczna metoda ze wszystkich trzech. Może ona jednak być przydatna podczas doraźnego uruchamiania pro-

gramów narzędziowych. Jak się wydaje, dla większości innych celów, plik konfiguracyjny XML jest najlepszym rozwiązaniem.

Uruchamianie przykładu Message

Skoro już zainstalowaliśmy Hibernate'a i bazę danych, a także stworzyliśmy nasz plik konfiguracyjny, pozostaje nam jedynie utworzenie kompletnych klas, a następnie skompilowanie i uruchomienie całości. W rozdziale pierwszym pominęliśmy najprostsze części wymaganych klas, zatem zamieszczamy je wszystkie w listingach 3.5, 3.6 oraz 3.7. Później przyjrzymy się niektórym szczegółom z nimi związanym.

Listing 3.5. *Klasa Message obiektów POJO*

```
package sample.entity;

public  class  Message  {
    private String message;

    public Message(String message) {
        this.message  =  message;
    }

    Message() {
    }

    public String getMessage() {
        return this.message;
    }

    public void setMessage(String message) {
        this.message  =  message;
    }
}
```

Listing 3.6 pokazuje prostą aplikację wypełniającą przykładami tabelę komunikatów.

Listing 3.6. *Kod tworzący przykładowy komunikat*

```
package sample;

import java.util.Date;

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

import sample.entity.Message;
```

```
public class PopulateMessages {
    public static void main(String[] args) {
            SessionFactory factory =
                new Configuration().configure().buildSessionFactory();
            Session session = factory.openSession();
            session.beginTransaction();

            Message m1 = new Message(
                "Hibernated a message on " + new Date());
            session.save(m1);
            session.getTransaction().commit();
            session.close();
    }
}
```

W końcu, listing 3.7 prezentuje pełen tekst aplikacji wymieniający wszystkie komunikaty w bazie danych.

Listing 3.7. *Aplikacja Message*

```
package sample;

import java.util.Iterator;
import java.util.List;

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

import sample.entity.Message;
public class ListMessages {
    public static void main(String[] args)
    {
        SessionFactory factory =
            new Configuration().configure().buildSessionFactory();
        Session session = factory.openSession();

        List messages = session.createQuery("from Message").list();
        System.out.println("Found " + messages.size() + " message(s):");

        Iterator i = messages.iterator();
        while(i.hasNext()) {
            Message msg = (Message)i.next();
            System.out.println(msg.getMessage());
        }
```

```
        session.close();
    }
}
```

Docelowe zadanie Anta `exportDDL` stworzy odpowiedni schemat w plikach bazy danych HSQLDB. Uruchomienie zadania `populateMessages` utworzy pole komunikatu (można je wielokrotnie przywoływać). Uruchomienie zadania `listMessages` wymieni komunikaty wpisane do tej pory do bazy danych.

■ **Uwaga** Ponieważ wybraliśmy opcję `drop="true"` dla podzadania `hbm2dll`, uruchomienie tego skryptu skutecznie usunie wszystkie dane w wymienionych tabelach. Rzadko kiedy zalecane jest uruchomienie takiego skryptu na maszynie posiadającej dostęp do bazy danych środowiska produkcyjnego ze względu na ryzyko przypadkowego usunięcia naszych danych!

Ustawiliśmy odpowiednie pola ścieżki klas w skrypcie kompilacji Anta. Aby uruchomić aplikację Hibernate'a, potrzebujemy plik `hibernate.jar` z jego katalogu głównego oraz podzbiór bibliotek dostarczonych w podkatalogu `lib`. Pochodzenie, przeznaczenie oraz dodatkowe opcje każdej z tych bibliotek są wyjaśnione w pliku tekstowym `README` umieszczonym w katalogu `lib`.

Większość pracy potrzebnej do uruchomienia tego przykładu to podstawowe drobiazgi konfiguracyjne wymagane przez każdą aplikację (napisanie skryptów Anta, ustawienie ścieżek klas, itd.). Na prawdziwą pracę składają się następujące zadania:

1. Utworzenie pliku konfiguracyjnego Hibernate'a
2. Utworzenie pliku odwzorowań
3. Zapisanie obiektów POJO (przedstawionych w rozdziale pierwszym)

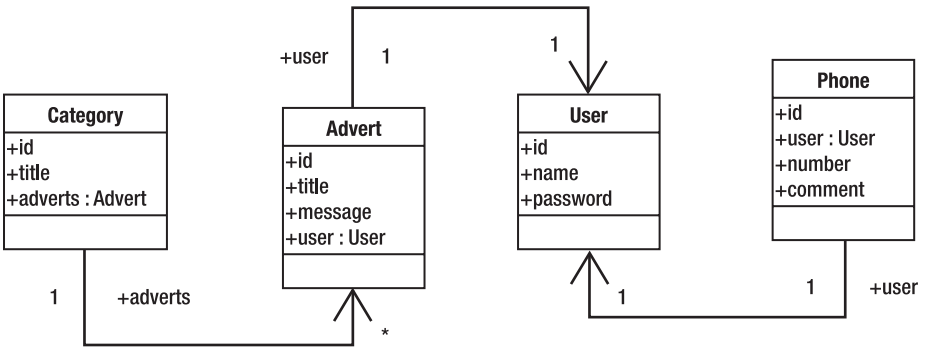
Utrwalanie wielu obiektów

Nasz przykład z rozdziału pierwszego był tak prostym scenariuszem utrwalania, jak tylko można sobie to wyobrazić.

W dalszej części tego rozdziału przedstawimy technologię utrwalania internetowej aplikacji reklamowej (pokazanej na rysunku3.1.).

Jest to duże uproszczenie kategorii klas wymaganych w tworzeniu aplikacji. Dla przykładu, nie uwzględniliśmy różnic pomiędzy rolami użytkowników aplikacji, ale powinno wystarczyć ukazanie niektórych prostszych związków pomiędzy klasami.

Szczególnie interesująca jest relacja typu „wiele do wielu" (ang. *many-to-many*) pomiędzy kategoriami i reklamami. Moglibyśmy chcieć posiadać wiele kategorii i reklam oraz móc umieścić jakąkolwiek reklamę w więcej niż jednej kategorii. Na przykład, pianino elektroniczne powinno znaleźć się zarówno w kategorii „Instrumenty", jak i w kategorii „Elektronika".



Rysunek 3.1. *Klasy internetowej reklamy*

Tworzenie trwałych klas

Zacznijmy od stworzenia obiektów POJO dla naszej aplikacji. Nie jest to niezbędnie konieczne w nowej aplikacji, gdyż mogą być one wygenerowane bezpośrednio z plików odwzorowań. Ponieważ jest to jednak już dobrze znany nam teren, może okazać się przydatne wprowadzenie pewnego kontekstu dla mającego nastąpić utworzenia plików odwzorowań.

Wiemy z diagramu klas, iż trzy klasy będą utrwalone w bazie danych (porównajmy listingi 3.8, 3.9 i 3.10). Każda klasa, która ma zostać utrwalona przez Hibernate'a, musi mieć domyślny konstruktor o zasięgu przynajmniej pakietu. Powinna też posiadać metody zwracające i ustawiające dla wszystkich atrybutów, które mają być utrwalone. Dostarczymy każdej z nich pole `id`, zezwalając, aby było ono kluczem podstawowym naszej bazy danych (lepsze jest korzystanie z zastępczych kluczy, gdyż zmiany w warstwie biznesowej mogą uczynić ryzykownym korzystanie z kluczy bezpośrednich).

■ **Uwaga** Klucz zastępczy jest to dowolna wartość (zazwyczaj liczbowa) o typie danych zależnym od liczby spodziewanych obiektów (np. 32-bitowy, 64-bitowy, etc.). Poza bazą danych klucz zastępczy nic nie znaczy – nie jest to ani numer klienta, ani numer telefonu, ani nic innego. Z tego względu, jeśli jakaś decyzja biznesowa duplikuje dotychczas niepowtarzalne dane, nie spowoduje to problemów, ponieważ dane biznesowe nie tworzą klucza podstawowego.

Oprócz domyślnego konstruktora dla każdej klasy, dostarczamy także konstruktor umożliwiający bezpośrednie przypisanie pól innych niż klucze podstawowe. Dzięki temu możemy stworzyć i wypełnić obiekt w jednej czynności zamiast kilku. Pozwólmy jednak, aby Hibernate zajął się alokacją naszych kluczy podstawowych.

Klasy pokazane na rysunku 3.1 są naszymi obiektami POJO. Ich implementacja jest ukazana w listingach: 3.8, 3.9 oraz 3.10.

Listing 3.8. *Klasa reprezentująca użytkowników*

```
package sample.entity;

public class User {
    private long id;
    private String name;
    private String password;

    public User(String name, String password) {
        this.name = name;
        this.password = password;
    }

    User() {
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    protected long getId() {
        return id;
    }

    protected void setId(long id) {
        this.id = id;
    }
}
```

Listing 3.9. *Klasa reprezentująca kategorie (każda z nich posiada powiązany zbiór obiektów typu* Advert*)*

```
package sample.entity;

import  java.util.HashSet;
import  java.util.Set;

public  class  Category  {
    private long id;
    private String title;
    private Set adverts = new HashSet();

    public Category(String title) {
        this.title = title;
        this.adverts  =  new  HashSet();
    }

    Category() {
    }

    public Set getAdverts() {
        return  adverts;
    }

    void setAdverts(Set adverts) {
        this.adverts  =  adverts;
    }

    public void addAdvert(Advert advert) {
        getAdverts().add(advert);
    }

    public String getTitle() {
        return  title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    protected long getId() {
        return id;
    }
```

```
    protected void setId(long id) {
        this.id  =  id;
    }
}
```

Listing 3.10. *Klasa reprezentująca reklamy (każdy egzemplarz posiada skojarzonego użytkownika, który zamieścił reklamę)*

```
package sample.entity;

public class Advert {
    private long id;
    private String title;
    private String message;
    private User user;

    public Advert(String title, String message, User user) {
        this.title = title;
        this.message = message;
        this.user = user;
    }

    Advert() {
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String getTitle() {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public User getUser() {
        return user;
    }
```

```
    public void setUser(User user) {
        this.user = user;
    }

    protected long getId() {
        return id;
    }

    protected void setId(long id) {
        this.id = id;
    }
}
```

Nie musieliśmy dodawać żadnych wyjątkowych funkcji do tych klas po to, aby obsługiwały narzędzie Hibernate. Mimo to zdecydowaliśmy się dostarczyć domyślne konstruktory o zasięgu pakietu w celu obsługi opcjonalnej funkcji Hibernate'a określanej jako „opóźnione ładowanie" (ang. *lazy loading*). Większość istniejących aplikacji będzie, „z pudełka" (ang. *out of the box*) zawierała obiekty POJO, które są kompatybilne z Hibernate'em.

Tworzenie odwzorowań obiektów

Skoro już mamy nasze obiekty POJO, musimy odwzorować je do bazy danych. Robimy to tworząc pośrednio lub bezpośrednio reprezentację pól każdego z nich jako wartości w kolumnach powiązanych tabel. Po kolei zajmujemy się każdym obiektem.

Zdefiniowana jest nazwa z pełnym kwalifikatorem odwzorowanego typu oraz tabela, w której chcielibyśmy go przechowywać (wykorzystaliśmy tabelę `aduser`, ponieważ `user` jest słowem kluczowym wielu baz danych).

Klasa posiada następujące trzy pola:

Pole `id`: Odpowiada kluczowi zastępczemu, używanemu i wygenerowanemu przez bazę danych. To specjalne pole jest obsługiwane przez element `<id>`. Nazwa pola jest określona przez atrybut name (aby `name="id"` zgadzało się z nazwą metody `getId`). Pole to jest typu `long`, ponieważ chcielibyśmy przechowywać jego wartości w kolumnach `long` bazy danych. Określamy, że powinno być ono generowane raczej przez bazę danych aniżeli przez Hibernate'a.

Pole `name`: Reprezentuje nazwę użytkownika. Powinno być przechowywane w kolumnie o nazwie `name`. Jest typu `String`. Nie zezwalamy na przechowywanie w kolumnie zdublowanych nazw.

Pole `password`: Reprezentuje hasło danego użytkownika. Powinno być przechowywane w kolumnie o nazwie password. Jest typu `String`.

Jeśli pamiętamy o powyższych cechach, przeanalizowanie pliku odwzorowań z listingu 3.11 powinno być wyjątkowo proste.

Listing 3.11. *Odwzorowanie klasy* `User` *do bazy danych*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="com.hibernatebook.chapter3.User" table="aduser">

        <id name="id" type="long" column="id">
            <generator class="native"/>
        </id>

        <property name="name" column="name" type="string" unique="true"/>

        <property name="password" column="password" type="string"/>

    </class>
</hibernate-mapping>
```

Odwzorowanie obiektu `Category` stanowi inny rodzaj relacji "wiele do wielu". Każdy obiekt typu `Category` jest skojarzony ze zbiorem reklam, podczas gdy dowolna reklama może być powiązana z wieloma kategoriami.

Element `<set>` informuje, że dane pole jest typu `java.util.Set`. W naszym przykładzie pole ma nazwę `adverts`. Ten rodzaj relacji wymaga utworzenia dodatkowej tabeli łączącej, dlatego określamy nazwę tabeli zawierającej tę informację.

Podajemy, że podstawowy klucz dla obiektów zawartych w tabeli `link`, którego używamy do pobierania elementów, jest reprezentowany przez kolumnę `id`. Określamy posiadającą pełen kwalifikator nazwę typu klasy umieszczonej w tabeli. W tabeli `link` wyszczególniamy kolumnę reprezentującą reklamy związane z każdą z kategorii.

I tym razem może wydawać się to skomplikowane, ale jeśli przyjrzymy się przykładowej tabeli z listingu 3.14, konieczność zamieszczenia każdego z pól w odwzorowaniach stanie się jasna (zob. listing 3.12).

Listing 3.12. *Odwzorowanie kategorii Class do bazy danych*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="sample.entity.Category" table="category">
```

```
    <id name="id" type="long" column="id">
        <generator class="native"/>
    </id>

        <property
                name="title"
                column="title"
                type="string"
                unique="true"/>

        <set name="adverts" table="link_category_advert" >
            <key column="category" foreign-key="fk_advert_category"/>
            <many-to-many class="sample.entity.Advert"
                column="advert"
                foreign-key="fk_category_advert"/>
        </set>

    </class>
</hibernate-mapping>
```

W końcu, reprezentujemy klasę `Advert` (zob. listing 3.13). Klasa ta przedstawia relację „wiele do jednego" (ang. *many-to-one*) – w tym przypadku jest to powiązanie z klasą `User`. Każda reklama musi należeć do pojedynczego użytkownika, ale dowolny użytkownik może zamieszczać wiele różnych reklam.

Listing 3.13. *Odwzorowanie klasy Advert do bazy danych*

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
    PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="sample.entity.Advert" table="advert">

        <id name="id" type="long" column="id">
            <generator class="native"/>
        </id>

        <property name="message" column="message" type="string"/>
        <property name="title" column="title" type="string"/>
        <many-to-one
            name="user"
            column="aduser"
            class="sample.entity.User"
            not-null="true"
```

```
            foreign-key="fk_advert_user"/>

    </class>
</hibernate-mapping>
```

Kiedy już stworzyliśmy poszczególne pliki odwzorowań, musimy przekazać Hibernate'owi, gdzie ma ich szukać. Jeżeli korzystamy z pliku konfiguracyjnego Hibernate'a (tak jak w przykładzie z rozdziału pierwszego), najprościej jest zamieścić od razu w nim odsyłacze do plików odwzorowań.

Dla potrzeb naszego przykładu bierzemy plik konfiguracyjny opisany w rozdziale pierwszym (listing 1.5) i dodajemy trzy poniższe zapisy odwzorowań zasobów:

```
<mapping resource="sample/entity/Advert.hbm.xml"/>
<mapping resource="sample/entity/Category.hbm.xml"/>
<mapping resource="sample/entity/User.hbm.xml"/>
```

Należy je zamieścić poniżej następującej linii:

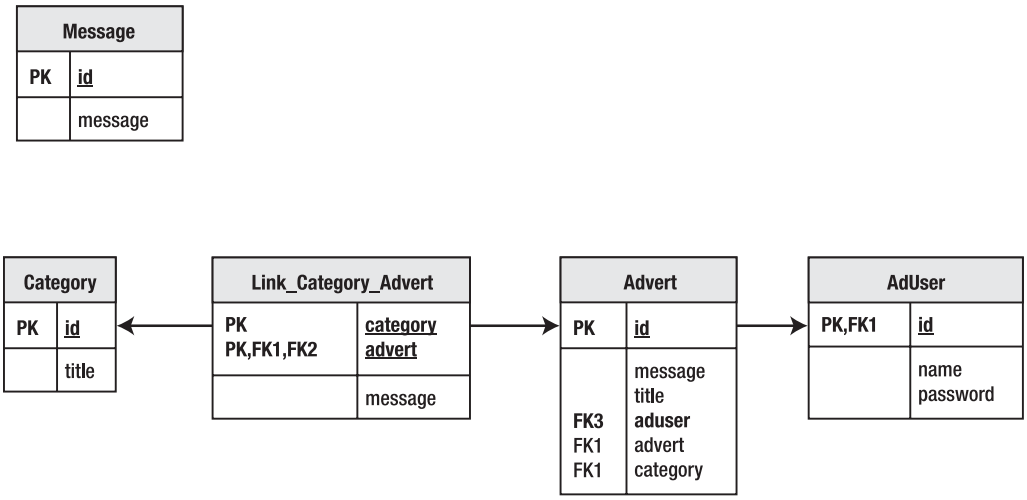
```
<mapping resource="sample/entity/Message.hbm.xml"/>
```

Ta część rozdziału może wydawać się zawiła, gdyż przypomina przelotną wizytę w dziedzinie odwzorowań i związanych z nimi pytań. W kolejnych rozdziałach dogłębnie przeanalizujemy odwzorowania – w rozdziale piątym ogólne zagadnienia odwzorowań, a w rozdziale siódmym pliki odwzorowań opartych na formacie XML. W szóstym rozdziale omówimy, w jaki sposób możemy wykorzystać nowe funkcje adnotacji Javy 5 do bezpośredniego reprezentowania odwzorowań w kodzie źródłowym.

Tworzenie tabel

Mając już za sobą etap odwzorowania obiektu oraz poprawnego ustawienia pliku konfiguracyjnego Hibernate'a, możemy już wygenerować skrypt tworzący bazę danych dla naszej aplikacji. Wywołujemy więc zadanie `exportDDL`, które kompiluje encje pokazane na rysunku 3.2.

Chociaż możemy bezpośrednio wygenerować bazę danych, zalecamy poświęcenie trochę czasu na wypróbowanie, jakiego oczekiwalibyśmy schematu wygenerowanego przez nasze odwzorowania. Pozwala to na sprawdzenie poprawności skryptu, aby upewnić się, że spełnia on nasze oczekiwania. Jeśli my i narzędzie przez nas używane zgodzimy się co do tego, jak sprawy mają wyglądać, wówczas wszystko jest w porządku. W przeciwnym razie, nasze odwzorowanie mogło być błędne lub mógł wystąpić drobny błąd w sposobie, w jaki powiązaliśmy nasze typy danych.



Rysunek 3.2. *Relacje pomiędzy encjami bazy danych*

Skrypt z listingu 3.14 został wygenerowany przez zadanie `exportDDL`. Łatwo można byłoby go wprowadzić samodzielnie i nietrudno jest porównać go z naszymi wcześniejszymi oczekiwaniami dotyczącymi schematu bazy danych (nieznacznie zmienione zostało formatowanie, ale pod każdym innym względem skrypt ten jest identyczny z wynikiem zadania).

Listing 3.14. *Skrypt wygenerowany przez zadanie* `exportDDL`

```
alter table advert
    drop constraint fk_advert_user;

alter table link_category_advert
    drop constraint fk_advert_category;

alter table link_category_advert
    drop constraint fk_category_advert;

drop table aduser if exists;
drop table advert if exists;
drop table category if exists;
drop table link_category_advert if exists;
drop table message if exists;

create table aduser (
    id bigint generated by default as identity (start with 1),
    name varchar(255),
    password varchar(255),
    primary key (id),
```

```
    unique (name));

create table advert (
    id bigint generated by default as identity (start with 1),
    message varchar(255),
    title varchar(255),
    aduser bigint not null,
    primary key (id));

create table category (
    id bigint generated by default as identity (start with 1),
    title varchar(255),
    primary key (id),
    unique (title));

create table link_category_advert (
    category bigint not null,
    advert bigint not null,
    primary key (category, advert));

create table message (
    id bigint generated by default as identity (start with 1),
    message varchar(255),
    primary key (id));

alter table advert
    add constraint fk_advert_user
    foreign key (aduser) references aduser;

alter table link_category_advert
    add constraint fk_advert_category
    foreign key (category) references category;

alter table link_category_advert
    add constraint fk_category_advert
    foreign key (advert) references advert;
```

Zwróćmy uwagę na ograniczenia obcych kluczy oraz na tabelę łączącą, która reprezentuje relację „wiele do wielu”.

Sesje

W rozdziale czwartym omówimy szczegółowo pełny cykl życia trwałych obiektów. Jeśli jednak chcemy stworzyć jakąkolwiek prostą aplikację Hibernate'a, musimy poznać podstawy dotyczące związku pomiędzy sesją i trwałymi obiektami.

Sesja i powiązane wzajemnie obiekty

Sesję tworzy się zawsze z obiektu `SessionFactory`. `SessionFactory` jest ciężkim obiektem (ang. *heavyweight object*) i zazwyczaj występuje jeden egzemplarz na aplikację. Pod pewnym względem przypomina narzędzie zarządzające pulą połączeń w podłączonej aplikacji. W aplikacji na platformie J2EE zwykle jest on pobierany jako zasób JNDI. Jest tworzony z obiektu `Configuration`, który z kolei pobiera informacje o konfiguracji Hibernate'a i wykorzystuje je do generowania odpowiedniego egzemplarza `SessionFactory`.

Sama sesja ma wiele wspólnego z obiektem JDBC Connection. Aby wczytać obiekt z bazy danych, musimy użyć sesji (pośrednio lub bezpośrednio). Przykładem takiego bezpośredniego użycia sesji byłoby (podobnie jak w rozdziale pierwszym) wywołanie metody `session.get()` bądź też utworzenie z sesji obiektu `Query` (wykazuje on duże podobieństwo do obiektu `PreparedStatement`).

Niebezpośrednie użycie sesji to na przykład wykorzystanie samego obiektu związanego z sesją. Jeśli pobraliśmy z bazy danych obiekt `Phone` używając sesji bezpośrednio, możemy pobrać obiekt `User` poprzez wywołanie metody `getUser()` – właściwej dla obiektu `Phone`. Możemy tak zrobić nawet jeśli skojarzony obiekt `User` nie został jeszcze załadowany (w wyniku opóźnionego wczytywania).

Obiekt, który nie został załadowany za pośrednictwem sesji może zostać wprost powiązany z sesją na kilka sposobów. Najprostszym z nich jest wywołanie metody `session.update()` przekazującej dany obiekt.

Sesja jednakże może zdziałać znacznie więcej – udostępnia pewne funkcje buforowania podręcznego, obsługuje opóźnione wczytywanie obiektów oraz monitoruje zmiany w powiązanych obiektach (tak, aby zmiany te mogły być utrwalone w bazie danych).

Transakcja Hibernate'a jest zazwyczaj wykorzystywana w podobny sposób do transakcji interfejsu JDBC. Posługujemy się nią w celu grupowania wzajemnie zależnych operacji Hibernate'a poprzez zezwolenie na zamykanie ich i wycofywanie transakcji w sposób atomowy. Służy ona również odizolowaniu operacji od zewnętrznych zmian w bazie danych. Hibernate może także skorzystać z opcji zakresu transakcji, aby ograniczyć niepotrzebną „gadatliwość" sterownika JDBC. Ustawia wówczas w kolejce wyrażenia SQL, aby były przesyłane razem, kiedy to możliwe na końcu transakcji.

Wszystko to omówimy dokładniej w rozdziale czwartym, a tymczasem wystarczy powiedzieć, że musimy zachować pojedynczy obiekt `SessionFactory` dla całej aplikacji. Dostęp do sesji powinien być jednakże możliwy tylko w pojedynczym wątku wykonawczym. Ponieważ sesja reprezentuje także informację buforowaną z bazy danych, wskazane jest zachowanie jej do wykorzystania w wątku aż do momentu, kiedy coś (zwłaszcza jakikolwiek wyjątek Hibernate'a) sprawi, że stanie się ona nieprawidłowa.

W listingu 3.15 przedstawiamy wzorzec, z którego mogą zostać utworzone obiekty dostępu do danych (DAO – ang. *Data Access Objects*). Oferuje on skuteczny sposób, w jaki wątek może pobrać i (jeśli to konieczne) stworzyć swoje sesje, nie wpływając znacząco na przejrzystość kodu.

Listing 3.15. *Klasa bazowa zarządzająca sesją w naszym przykładzie*

```
package sample.dao;

import java.util.logging.Level;
import java.util.logging.Logger;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class DAO {

    protected DAO() {
    }

    public static Session getSession() {
        Session session = (Session) DAO.session.get();
        if (session == null) {
            session = sessionFactory.openSession();
            DAO.session.set(session);
        }
        return session;
    }

    protected void begin() {
        getSession().beginTransaction();
    }

    protected void commit() {
        getSession().getTransaction().commit();
    }

    protected void rollback() {
        try {
            getSession().getTransaction().rollback();
        } catch( HibernateException e ) {
            log.log(Level.WARNING,"Cannot rollback",e);
        }
        try {
            getSession().close();
        } catch( HibernateException e ) {
            log.log(Level.WARNING,"Cannot close",e);
        }
        DAO.session.set(null);
    }
```

```
    public static void close() {
        getSession().close();
        DAO.session.set(null);
    }

    private static final Logger log = Logger.getAnonymousLogger();

    private static final ThreadLocal session = new ThreadLocal();

    private static final SessionFactory sessionFactory =
        new Configuration().configure().buildSessionFactory();
}
```

Korzystanie z sesji

Tworzenie i usuwanie obiektów POJO to najpowszechniejsze przypadki ich użycia. W obu sytuacjach chcemy, aby zmiany były odzwierciedlone w bazie danych.

Dla przykładu, chcemy mieć możliwość stworzenia użytkownika poprzez określenie nazwy użytkownika i hasła. Chcielibyśmy także przechowywać potem te informacje w bazie danych.

Logika służąca utworzeniu użytkownika (i odzwierciedleniu tego faktu w bazie danych) jest niewiarygodnie prosta, co pokazano w listingu 3.16.

Listing 3.16. *Tworzenie obiektu* `User` *i odzwierciedlenie tego w bazie danych*

```
try {
    begin();
    User user = new User(username,password);
    getSession().save(user);
    commit();
    return user;
} catch( HibernateException e ) {
    rollback();
    throw new AdException("Could not create user " + username,e);
}
```

Rozpoczynamy transakcję, tworzymy nowy obiekt `User`, zapisujemy go w sesji i zatwierdzamy transakcję. Jeżeli napotykamy na jakiś problem (np. gdy encja `User` o tej nazwie użytkownika już wcześniej została utworzona w bazie danych), wtedy zostanie wyrzucony wyjątek Hibernate'a, a cała transakcja będzie wycofana.

Aby nauczyć się pobierania obiektu `User` z bazy danych, musimy najpierw zrobić pierwszą dygresję na temat języka HQL. Przypomina on do pewnego stopnia język SQL, ale musimy pamiętać, że HQL odnosi się raczej do nazw wykorzystywanych w plikach odwzorowań aniżeli nazw tabel i kolumn odnośnej bazy danych.

Odpowiednie zapytanie HQL pobierające użytkowników o danej nazwie wygląda następująco:

```
from User where name= :username
```

gdzie `User` jest nazwą klasy, a `:username` jest nazwanym parametrem HQL, któremu wartość zapewni nasz kod podczas przeprowadzania kwerendy. Jest to bardzo podobne do języka SQL ze względu na gotowe wyrażenie służące do osiągnięcia tego samego celu:
`select * from user where name = ?`

Listing 3.17 przedstawia cały kod potrzebny do pobrania użytkownika ze względu na konkretną jego nazwę.

Listing 3.17. *Pobieranie obiektu* `User` *z bazy danych*

```
try {
    begin();
    Query q = getSession().createQuery("from User where name = :username");
    q.setString("username",username);
    User user = (User)q.uniqueResult();
    commit();
    return user;
} catch( HibernateException e ) {
    rollback();
    throw new AdException("Could not get user " + username,e);
}
```

Rozpoczynamy transakcję, tworzymy obiekt `Query` (o podobnym przeznaczeniu jak `PreparedStatement` w podłączonych aplikacjach), przypisujemy parametrowi kwerendy właściwą nazwę użytkownika, a następnie umieszczamy w spisie wyniki zapytania. Wydobywamy użytkownika (jeśli pomyślnie został pobrany) i zatwierdzamy transakcję. Jeśli wystąpi problem z odczytem danych, transakcja zostanie wycofana.

Oto kluczowa linia używana do uzyskiwania encji `User`:

```
User user = (User)q.uniqueResult();
```

Wykorzystujemy metodę `uniqueResult()`, ponieważ zapewnia ona, że wyjątek zostanie wyrzucony, jeśli nasze zapytanie zidentyfikuje więcej niż jeden obiekt `User` dla podanej nazwy użytkownika. W zasadzie, może się tak zdarzyć, jeżeli ograniczenia odnośnej bazy danych nie zgadzają się z ograniczeniami naszego odwzorowania w zakresie pola unikatowej nazwy użytkownika, a wyjątek jest odpowiednim sposobem obsługi tego błędu.

Logika odpowiedzialna za usunięcie użytkownika z bazy danych (listing 3.18) jest jeszcze prostsza niż ta związana z utworzeniem go.

Listing 3.18. *Usuwanie obiektu* `User` *i odzwierciedlanie tego w bazie danych*

```
try {
    begin();
    getSession().delete(user);
    commit();
} catch( HibernateException e ) {
    rollback();
    throw new AdException("Could not delete user " + user.getName(),e);
}
```

Po prostu nakazujemy sesji usunięcie obiektu `User` z bazy danych oraz zatwierdzamy transakcję. Jeżeli wystąpi problem, transakcja zostanie wycofana – np. gdy użytkownik został już wcześniej usunięty.

Zapoznaliśmy się ze wszystkimi podstawowymi operacjami przeprowadzanymi na naszych danych, przyjrzymy się zatem architekturze, której będziemy używać.

Tworzenie obiektów dostępu do danych (DAO)

Wzorzec DAO jest dobrze znany większości programistów. Celem jest tutaj oddzielenie obiektów typu POJO od logiki wykorzystywanej do ich utrwalania w bazie danych i ich pobierania z niej. Szczegóły implementacji różnią się – w krańcowym przypadku, obiekty te mogą być udostępnione jako egzemplarze interfejsów stworzone z klasy fabryki. Umożliwiają one istnienie całkowicie przełączalnej warstwy bazy danych. Na potrzeby naszego przykładu wybraliśmy grupę konkretnych klas DAO. Każda klasa DAO reprezentuje operacje, które można wykonać na obiektach typu POJO.

Omówiliśmy już klasę bazową DAO z listingu 3.15, a poprzednie przykłady już te informacje wykorzystywały.

Aby wspomóc proces hermetyzacji szczegółów operacji przeprowadzanych na bazie danych, przechwytujemy którykolwiek wyrzucony wyjątek `HibernateException`. Następnie opakowujemy go w egzemplarz `AdException`, tak jak pokazano w listingu 3.19.

Listing 3.19. *Klasa* `AdException` *z naszego przykładu*

```
package sample;

public class AdException extends Exception {
    public AdException(String message) {
        super(message);
    }

    public AdException(String message, Throwable cause) {
        super(message,cause);
    }
}
```

`UserDAO` udostępnia wszystkie metody wymagane do utworzenia, pobrania i usunięcia obiektów typu `User` (zob. listing 3.20). Zmiany wprowadzone do tego obiektu zostaną utrwalo-

ne w bazie danych pod koniec transakcji.

Listing 3.20. *Klasa* `UserDAO` *z naszego przykładu*

```
package sample.dao;

import org.hibernate.HibernateException;
import org.hibernate.Query;

import sample.AdException;
import sample.entity.User;

public class UserDAO extends DAO {
    public UserDAO() {
    }
    public User get(String username)
        throws AdException
    {
        try {
            begin();
            Query q = getSession().createQuery("from User where name = :username");
            q.setString("username",username);
            User user = (User)q.uniqueResult();
            commit();
            return user;
        } catch( HibernateException e ) {
            rollback();
            throw new AdException("Could not get user " + username,e);
        }
    }

    public User create(String username,String password)
        throws AdException
    {
        try {
            begin();
            User user = new User(username,password);
            getSession().save(user);
            commit();
            return user;
        } catch( HibernateException e ) {
            rollback();
            throw new AdException("Could not create user " + username,e);
        }
    }
```

```
    public void delete(User user)
        throws AdException
    {
        try {
            begin();
            getSession().delete(user);
            commit();
        } catch( HibernateException e ) {
            rollback();
            throw new AdException("Could not delete user " + user.getName(),e);
        }
    }
}
```

CategoryDAO udostępnia wszystkie metody wymagane do utworzenia, pobrania i usunięcia obiektów typu Category (zob. listing 3.21). Zmiany wprowadzone do tego obiektu zostaną utrwalone w bazie danych pod koniec transakcji.

Listing 3.21. *Klasa* CategoryDAO *z naszego przykładu*

```
package sample.dao;

import java.util.List;

import org.hibernate.HibernateException;
import org.hibernate.Query;

import sample.AdException;
import sample.entity.Category;

public class CategoryDAO extends DAO {
    public Category get(String title) throws AdException {
        try {
            begin();
            Query q = getSession().createQuery(
                "from Category where title = :title");
            q.setString("title", title);
            Category category = (Category) q.uniqueResult();
            commit();
            return category;
        } catch (HibernateException e) {
            rollback();
            throw new AdException("Could not obtain the named category " + title, e);
        }
    }
```

```
    public List list() throws AdException {
        try {
            begin();
            Query q = getSession().createQuery("from Category");
            List list = q.list();
            commit();
            return list;
        } catch (HibernateException e) {
            rollback();
            throw new AdException("Could not list the categories", e);
        }
    }

    public Category create(String title) throws AdException {
        try {
            begin();
            Category cat = new Category(title);
            getSession().save(cat);
            commit();
            return null;
        } catch (HibernateException e) {
            rollback();
            throw new AdException("Could not create the category", e);
        }
    }

    public void save(Category category) throws AdException {
        try {
            begin();
            getSession().update(category);
            commit();
        } catch (HibernateException e) {
            rollback();
            throw new AdException("Could not save the category", e);
        }
    }

    public void delete(Category category) throws AdException {
        try {
            begin();
            getSession().delete(category);
            commit();
        } catch (HibernateException e) {
            rollback();
            throw new AdException("Could not delete the category", e);
        }
```

```
    }
}
```

AdvertDAO udostępnia wszystkie metody wymagane do utworzenia lub usunięcia obiektów typu Advert (reklamy są zawsze pobierane poprzez wybranie ich spośród kategorii, dlatego więc są pośrednio wczytywane przez klasę CategoryClass). Zmiany wprowadzone do tego obiektu zostaną utrwalone w bazie danych pod koniec transakcji (zob. listing 3.22).

Listing 3.22. *Klasa* AdvertDAO *z naszego przykładu*

```
package sample.dao;

import org.hibernate.HibernateException;

import sample.AdException;
import sample.entity.Advert;
import sample.entity.User;

public class AdvertDAO extends DAO {
    public Advert create(String title, String message, User user)
            throws AdException {
        try {
            begin();
            Advert advert = new Advert(title, message, user);
            getSession().save(advert);
            commit();
            return advert;
        } catch (HibernateException e) {
            rollback();
            throw new AdException("Could not create advert", e);
        }
    }

    public void delete(Advert advert)
        throws AdException
    {
        try {
            begin();
            getSession().delete(advert);
            commit();
        } catch (HibernateException e) {
            rollback();
            throw new AdException("Could not delete advert", e);
        }
    }
}
```

Jeśli porównamy ilość kodu wymaganego tutaj do stworzenia naszych klas DAO z ilością kodu potrzebnego do ich implementacji przy użyciu zwykłego podejścia JDBC, zauważymy bez trudu, że logika Hibernate'a jest skondensowana w sposób godny podziwu.

Przykładowy klient

Listing 3.23 pokazuje kod z naszego przykładu podający wszystko razem. Oczywiście, nie jest to cała aplikacja, ale mamy już teraz wszystkie obiekty DAO konieczne do zarządzania bazą danych obsługującą reklamy. Przykład ten daje przedsmak tego, w jaki sposób mogą być one użyte.

Kod powinien być uruchamiany poprzez zadania skryptu narzędzia Ant podane w listingu 3.1. Po wykonaniu zadania exportDDL tworzącego pustą bazę danych, powinniśmy uruchomić zadania `createUsers` i `createCategories`, aby dostarczyć pierwszych użytkowników i pierwsze kategorie. Następnie uruchamiamy zadanie `postAdverts` zamieszczające reklamy w bazie danych. Na koniec wykonujemy zadanie `listAdverts` wyświetlające zapisane dane.

Kod wywołujący obiekty DAO do wykonania powyższych zadań podano w listingu 3.23.

Listing 3.23. *Klasa tworząca użytkowników w naszym przykładzie*

```
package sample;

import sample.dao.DAO;
import sample.dao.UserDAO;

public class CreateUser {
    public static void main(String[] args) {

        if (args.length != 2) {
            System.out.println("params required: username, password");
            return;
        }
        String username = args[0];
        String password = args[1];

        try {
            UserDAO userDao = new UserDAO();
            System.out.println("Creating user " + username);
            userDao.create(username, password);
            System.out.println("Created user");
            DAO.close();
        } catch (AdException e) {
            System.out.println(e.getMessage());
        }
    }
}
```

Klasa `CreateUser` używa klasy `UserDAO` do stworzenia i utrwalenia odpowiedniego obiektu typu `User`. Szczegóły dotyczące utworzonych (dwóch) użytkowników są zaczerpnięte z parametrów wiersza poleceń udostępnionych w zadaniu Anta `createUsers`.

W listingu 3.24 tworzymy obiekty `Category` za pośrednictwem klasy `CategoryDAO`. I tym razem czerpiemy szczegóły z wiersza poleceń skryptu Anta.

Listing 3.24. *Klasa tworząca kategorie w naszym przykładzie*

```
package sample;

import sample.dao.CategoryDAO;
import sample.dao.DAO;

public class CreateCategory {
    public static void main(String[] args) {

        if (args.length != 1) {
            System.out.println("param required: categoryTitle");
            return;
        }

        CategoryDAO categories = new CategoryDAO();
        String title = args[0];
        try {
            System.out.println("Creating category " + title);
            categories.create(title);
            System.out.println("Created category");
            DAO.close();
        } catch (AdException e) {
            System.out.println(e.getMessage());
        }

    }
}
```

Kod z listingu 3.25 umożliwia nam utworzenie reklamy dla wcześniej istniejącego użytkownika we wcześniej istniejącej kategorii. Zwróćmy uwagę, w jaki sposób wykorzystaliśmy klasy `UserDAO` i `CategoryDAO` do otrzymania obiektów typu `User` i `Category` z bazy danych. Tak jak w przypadku użytkownika i kategorii, szczegóły dotyczące reklam podaje zadanie narzędzia Ant.

Listing 3.25. *Klasa tworząca reklamy w naszym przykładzie*

```
package sample;

import sample.dao.AdvertDAO;
```

```
import sample.dao.CategoryDAO;
import sample.dao.DAO;
import sample.dao.UserDAO;
import sample.entity.Advert;
import sample.entity.Category;
import sample.entity.User;
public class PostAdvert {
    public static void main(String[] args) {

        if (args.length != 4) {
System.out.println("params required: username, categoryTitle, title, message");
            return;
        }
        String username = args[0];
        String categoryTitle = args[1];
        String title = args[2];
        String message = args[3];

        try {
            UserDAO users = new UserDAO();
            CategoryDAO categories = new CategoryDAO();
            AdvertDAO adverts = new AdvertDAO();

            User user = users.get(username);
            Category category = categories.get(categoryTitle);
            Advert advert = adverts.create(title, message, user);

            category.addAdvert(advert);
            categories.save(category);

            DAO.close();
        } catch (AdException e) {
            e.printStackTrace();
        }
    }
}
```

Wreszcie, w listingu 3.26 używamy klasy `CategoryDAO` do iteracyjnego przeglądania kategorii oraz zamieszczonych wewnątrz nich reklam pobranych z bazy danych. Łatwo można zrozumieć, jak ta logika może być teraz włączona do pliku JSP lub serwletu. Może być nawet wykorzystana przez komponenty sesji EJB.

Listing 3.26. *Klasa wyświetlająca zawartość bazy danych*

```
package sample;

import java.util.Iterator;
import java.util.List;

import sample.dao.CategoryDAO;
import sample.dao.DAO;
import sample.entity.Advert;
import sample.entity.Category;

public class ListAdverts {
    public static void main(String[] args) {
        try {
            List categories = new CategoryDAO().list();
            Iterator ci = categories.iterator();
            while(ci.hasNext()) {
                Category category = (Category)ci.next();
                System.out.println("Category: " + category.getTitle());
                System.out.println();
                Iterator ai = category.getAdverts().iterator();
                while(ai.hasNext()) {
                    Advert advert = (Advert)ai.next();
                    System.out.println();
                    System.out.println("Title: " + advert.getTitle());
                    System.out.println(advert.getMessage());
                    System.out.println(" posted by " + advert.getUser().getName());
                }
            }

            DAO.close();
        } catch( AdException e ) {
            System.out.println(e.getMessage());
        }

    }
}
```

Duża część powyższej logiki jest albo informacją wyjścia, albo informacją dotyczącą uzyskania dostępu do samych kolekcji. Miłośnicy Javy 5 dostrzegą z pewnością oczywistą możliwość wykorzystania w tym przykładzie ogólnych i zaawansowanych pętli typu `for`. Listing 3.27 oferuje przedsmak uproszczonej wersji kodu.

Listing 3.27. *Ulepszanie obiektów DAO dzięki możliwościom Javy 5*

```
List<Category> categories = new CategoryDAO().list();
for( Category category : categories ) {
    // ...
    for( Advert advert : category.getAdverts() ) {
        // ...
    }
}
DAO.close();
```

Podczas uruchamiania tych przykładowych aplikacji mogliśmy zauważyć, że logujący interfejs API i narzędzie Ant wykazują się sporą "gadatliwością". Duża część tych komunikatów może zostać objęta kontrolą lub wyeliminowana w aplikacji produkcyjnej.

Możemy także zauważyć, że ponieważ uruchamiamy każdą z tych aplikacji jako nowe zadanie (kilkakrotnie w przypadku zadań tworzących dane), zadania realizowane są stosunkowo powoli. Jest to wynikiem wielokrotnego tworzenia „ciężkiego" obiektu `SessionFactory`. Powstaje on z każdego wywołania wirtualnej maszyny Javy (JVM), które jest przeprowadzane z zadania typu java task narzędzia Ant. Nie jest to zatem problem „rzeczywistych" aplikacji.

Podsumowanie

W tym rozdziale nauczyliśmy się:

- w jaki sposób opanować obsługę narzędzi Hibernate'a,
- jak utworzyć i uruchomić przykład z rozdziału pierwszego,
- jak utworzyć od podstaw nieco większą aplikację, generując tabelę bazy danych przy użyciu zadania Anta `hbm2ddl`.

Wszystkie pliki opisane zarówno w tym rozdziale, jak i w pozostałych, można pobrać ze strony internetowej wydawnictwa Apress (`www.apress.com`).

W następnym rozdziale przyjrzymy się architekturze Hibernate'a oraz cyklowi życia aplikacji na nim opartych.