Perzistencija objekata u Javi: Hibernate

Kurs Programski sistemi, Univerzitet Singidunum

Prilagođeno iz: Christian Bauer, Gavin King, *Java Persistence with Hibernate*, Manning, 2007

Tabelarno predstavljanje podataka u relacionom sistemu je fundamentalno različito od mreže objekata koja se koristi u objektno orijentisanim Java aplikacijama. Hibernate je alat koji premošćava jaz između relacionih i objektno orijentisanih tehnologija (engl. *paradigm mismatch* ili *impedance mismatch*), odnosno služi za objektno-relaciono preslikavanje (ORM, object-relational mapping). Hibernate taj zadatak obavlja na veoma praktičan, realističan i direktan način.

Prva verzija alata Hibernate pojavila se 2001. kada je njegov autor, Gevin King, zaključio da postojeći alati nisu dorasli složenim šemama podataka u aplikacijama. Tada je Hibernate započeo svoj život kao nekomercijalni, open-source projekat. Godine 2003. pridružio se porodici jboss.org i sada za njega postoji i komercijalna podrška, ali srećom po programere, i dalje je besplatan alat otvorenog koda.

U međuvremenu, podrška za perzistenciju u JaviEE (trenutno u verziji 7), tzv. JPA (Java Persistence API, trenutno u verziji 2.1) je napredovala. Hibernate tim je aktivno učestvovao u razvoju novog standarda; važna odluka koja je tom prilikom doneta je bila da se standardizuje ono što radi u praksi, a tu je Hibernate blistao. Kada je reč o specifikaciji JPA u JavaEE, može se reći da je Hibernate njena implementacija, ali treba imati u vidu da Hibernate nudi i neke funkcionalnosti koje nisu standardizovane, tj. predstavlja "nadskup" JPA.

Šta je perzistencija?

Gotovo svim aplikacijama potrebno je perzistiranje podataka. U Javi perzistencija obično podrazumeva čuvanje podataka u relacionoj bazi podataka uz korišćenje SQL sintakse. Relacione baze podataka su ušle u široku upotrebu zato što su veoma prilagodljiv i robustan način za upravljanje podacima. Zbog kompletne i dosledne teorijske osnove relacionog modela, relacione baze između ostalog efikasno garantuju i štite integritet podataka.

Sistemi za upravljanje bazama podataka nisu specifični za Javu, niti su relacione baze specifične za određenu aplikaciju. Taj važan princip zove se nezavisnost podataka. Drugim rečima, *podaci traju duže od bilo koje aplikacije*. Relaciona tehnologija omogućuje razmenu podataka između različitih podataka,

ili između različitih tehnologija koje čine delove iste aplikacije. Relaciona tehnologija je zajednički imenilac mnogih različitih sistema i tehnoloških platformi i sreće se u gotovo svim poslovnim aplikacijama.

Da bi se efikasno koristio Hibernate, neophodno je solidno razumevanje relacionog modela i jezika SQL. Prisetimo se zato nekih SQL termina i naredbi.

Kreiranje tabela

```
CREATE TABLE muzicari(muzicar_id INT, prezime CHAR(40),
ime CHAR(40), nadimak CHAR(40))
```

Unos podataka u tabele

```
INSERT INTO muzicari(muzicar_id, prezime, ime, nadimak) VALUES
(2, 'Lydon', 'John', 'Johnny Rotten')
```

Izmena zapisa koji su već uneti u tabelu

```
UPDATE albumi SET godina = 1994 WHERE album_id = 4
```

Brisanje zapisa iz tabele

```
DELETE from albumi WHERE album_id = 4
```

Upiti

```
SELECT naslov FROM albumi WHERE categorija = 'jazz'
```

Ukrštanje (join)

```
SELECT grupe.ime_grupe FROM grupe, albumi WHERE
albumi.kategorija = 'alternativa' AND grupe.grupa_id = albumi.grupa_id
```

Podupiti

```
SELECT naslov FROM albumi, WHERE grupa_id IN
(SELECT grupe.grupa_id FROM grupe, grupa_muzicar WHERE
grupa_muzicar.muzicar_id = 2 AND
grupe.grupa_id = grupa_muzicar.grupa_id)
```

Transakcija je jedna ili više SQL naredbi prema kojima se treba odnositi kao prema nedeljivoj jedinici rada. Ako se bilo koja od naredbi koje čine transakciju ne izvrši kako treba, od cele transakcije mora da se odustane, uključujući i poništavanje svih naredbi koje su eventualno uspešno izvršene pre naredbe koja nije uspela. Ako se uspešno izvrši ceo niz naredbi koje čine transakciju, bazi podataka se šalje signal da efekte transakcije učini trajnim; ta operacija se zove *commit*. Odustajanje od transakcije se zove *rollback*.

Relaciona baza je jedna strana alata za objektno-relaciono preslikavanje; druga strana sastoji se od objekata u Java aplikaciji koji treba da se perzistiraju i učitaju iz baze podataka pomoću jezika SQL. Postoje dva rešenja za ORM problem kojima ćemo posvetiti pažnju: serijalizacija i objektno orijentisane baze podataka (OODBMS).

Serijalizacija

U Javu je ougrađen mehanizam za perzistenciju: to je serijalizacija. Ona omogućuje "snimanje" stanja mreže objekata (stanja aplikacije) u tok bajtova, koji se zatim može snimiti u datoteku ili smestiti u bazu podataka. Problem sa serijalizacijom je u tome što se serijalizovanoj mreži povezanih objekata može pristupiti samo kao celini, tj. nije moguće iščitati podatke iz toka bez njegovog prethodnog kompletnog deserijalizovanja. To znači da se serijalizovani tok bajtova ne može koristiti za pretragu ili agregaciju velikih skupova podataka. Nije moguće čak ni pristupiti pojedinačnim objektima ili podskupovima objekata. Imajući u vidu tekuću tehnologiju, serijalizacija nije odgovarajuća kao rešenje za perzistenciju konkurentnih Web aplikacija i može se razmatrati samo u slučaju prostijih desktop programa.

Objektno orijentisane baze podataka

Pošto Java radi sa objektima, bilo bi idealno kada bi oni mogli da se sačuvaju u bazi podataka bez ikakvog menjanja ili kalupljenja objektnog modela. Sredinom devedesetih godina objektno orijentisane baze podataka su postale popularne, a zasnivaju se na ideji da se u njima sačuva mreža objekata, sa svim čvorovima i pokazivačima, tako da ta mreža objekata može da se rekonstruiše u memoriji. Objektno orijentisan DBMS je u stvari više proširenje okruženja aplikacije nego samostalan sistem za čuvanje podataka. OODBMS je obično implementiran u više slojeva, sa tesnom vezom između delova za smeštanje podataka, privremene memorije za objekte i klijentskom aplikacijom koje komuniciraju pomoću namenskog mrežnog protokola. Objektno orijentisane baze podataka nisu postale popularne i većina programera radije radi sa relacionim bazama. Razlog za to je i činjenica da je objektno orijentisana baza vezana za aplikaciju, tj. nije nezavisan sistem za čuvanje podataka kao što je to relaciona baza (tj. podaci sačuvani u bazi nisu nezavisni od aplikacije).

JDBC

Kada se radi sa SQL bazom podataka u Java aplikaciji, Java kod generiše SQL naredbe bazi podataka preko API-ja Java Database Connectivity (JDBC). Bez obzira da li su SQL naredbe ručno napisane u Java kodu, ili se usput generišu, JDBC API se koristi za povezivanje argumenata u pripremi parametara za upit, izvršavanje upita, pregled tabele sa rezultatima upita, dobijanje vrednosti iz skupa rezultata i sl.

Da bismo zaista razumeli svrhu ORM alata, podsetićemo se ukratko kako se radi sa bazama podataka u Javi preko interfejsa JDBC, odnosno kako se ovaj interfejs koristi za pristup alatima za baze podataka (DBMS, Database Management Systems).

Da bi se JDBC upotrebio u programu, mora se navesti uvesti paket `java.sql`.

JDBC objekti najnižeg nivoa pripadaju klasi `Connection`. Oni su stvarna veza sa sistemom za rad sa bazama podataka (DBMS) i kreiraju se sledećom sintaksom:

```
String url = "jdbc:mysql://localhost:3306";
String username = “mysql”;
String password = “sa”;
Class.forName("com.mysql.jdbc.Driver");
Connection connection = null;
try {
    connection = DriverManager.getConnection(url, username, pasword);
} catch (Exception ex) {
    ex.printStackTrace();
} finally {
    if( con != null ) {
        try {
            con.close( );
        } catch( Exception e ) {
            e.printStackTrace( );
        }
    }
}
```

Prva naredba, `Class.forName()`, registruje JDBC drajver koji će se koristiti. U try-catch bloku pravi se objekat veze sa bazom.

Objekti klase `Statement` predstavljaju pojedinačne SQL naredbe koje će se izdavati objektu `Connection`:

```
Statement stat = connection.prepareStatement
    ("SELECT * FROM tblWORKLOAD");
```

Posebna vrsta naredbi su tzv. pripremljene naredbe (prepared statement). Te naredbe svoj SQL kod čuvaju već kompajliran da bi se brže izvršavao. Pripremljene naredbe su objekti klase `PreparedStatement`. Sledeći kod priprema SQL naredbu za prikaz zapisa tabele:

```
SELECT * FROM tblWORKLOAD WHERE status = ?
PreparedStatement prep = connection.prepareStatement
    ("SELECT * FROM tblWORKLOAD WHERE status = ?");
prep.setString(1,"W");
```

Pripremljena naredba se generiše pomoću objekta klase `Connection`. Znak pitanja je mesto za parametar koji se prosleđuje upitu; da bi se on dostavio, koristi se metod `setString()` objekta `PreparedStatement` koji služi za prosleđivanje stringa "W" kao prvog (i u ovom slučaju jedinog) parametra upita (postoje i druge metode za različite tipove parametara, npr. `setAsciiStream()`, `setBigDecimal()`, `setBinaryStream()`, `setBoolean()`, `setByte()`, `setBytes()`, `setDate()`, `setDouble()`, `setFloat()`, `setInt()`, `setLong()`, `setNull()`, `setObject()`, `setShort()`, `setString()`, `setTime()`, `setTimestamp()` i `setUnicodeStream()`). Kada u SQL stringu ima više znakova pitanja, numeracija parametara i njihov tip mora da se poklopi sa njihovim rasporedom u SQL stringu.

SQL naredbe SELECT kao što je prethodna vraćaju podatke koji se smeštaju u skup rezultata; za to služi JDBC klasa `ResultSet` i odgovarajući metod `executeQuery()`. Ako se koristi neka SQL naredba koja ne vraća rezultate (INSERT, DELETE ili UPDATE), onda treba upotrebiti metod `executeUpdate()` klase `Statement`. Na primer,

```
ResultSet rs = prep.executeQuery();
while(rs.next() ) {
    System.out.println( rs.getString("URL"));
}
```

Kada se poziva metod `executeQuery()` za pripremljenu naredbu kao u prethodnom primeru, za skup rezultata rs treba pozvati metod `next()` da bi se dobio prvi zapis iz rezultata. Metod `next()` će vratiti false kada više ne bude bilo rezultata. U sledećem redu, koristi se metod `getString()` za ispis polja iz tražene tabele. String URL u ovom redu zadaje ime polja koje treba da se iščita iz dobijenog skupa rezultata upita. Ako SQL naredba ne vraća rezultate, postupak je malo drugačiji. U tom slučaju treba koristiti metod `executeUpdate()` umesto `executeQuery()` jer nema povratne vrednosti, već se samo izvršava SQL.

```
prep = connection.prepareStatement("DELETE * FROM tblWORKLOAD
    WHERE status = ?");
prep.setString(1,"W");
prep.executeUpdate();
```

Aplikacija ne radi direktno sa tabelarnom predstavom poslovnih entiteta, već ima sopstveni, objektno orijentisani model entiteta. Umesto da direktno radi sa poljima i zapisima SQL skupa rezultata, poslovna logika Java aplikacije komunicira sa bazom i "pretvara" je u skup povezanih objekata.

Čemu služi ORM alat?

ORM alat služi za automatsku (i transparentnu) perzistenciju objekata Java aplikacije u bazi podataka, a to se postiže pomoću metapodataka koji opisuju preslikavanje između objekata i baze. ORM alat zapravo transformiše podatke

iz jedne predstave u drugu, u oba smera, što naravno utiče na performanse aplikacije. Međutim, ORM alat je implementiran kao middleware i nudi mnogo načina za optimizaciju koji ne postoje u slojevima za perzistenciju koji se prave ručno.

ORM rešenje se sastoji iz sledeća četiri dela:

- API za izvođenje osnovnih CRUD (create-read-update-delete) operacija nad objektima perzistiranih klasa
- Jezik ili API za zadavanje upita koji rade sa klasama i svojstvima klasa
- Uslužnog alata za zadavanje metapodataka za preslikavanje
- Tehnike za interakciju sa transakcionim objektima koji izvode proveru da li je prezistirani objekat izmenjen (engl. *dirty checking*), "lenjo" dohvatanje kolekcija (engl. *lazy fetching*) i sl.

Termin kompletan ORM (engl. full ORM) se koristi za svaki alat koji je u stanju da automatski generiše SQL na osnovu metapodataka. ORM alat je prilično složen - manje od servera aplikacija, na primer, ali složeniji od frejmvorka kao što su Tapestry ili Struts. Prednosti korišćenja ORM alata su sledeće:

- Povećanje produktivnosti. Kod koji obavlja perzistenciju je najdosadniji kod u Java aplikaciji. Uz Hibernate taj kod nije potrebno pisati, tj. programeri mogu da se posvete problemu koji zaista rešavaju.
- Lakše održavanje koda. Manje koda čini sistem razumljivijim, jer se taj kod odnosi isključivo na problem koji se rešava. Takođe, uz ORM alat kod je više objektno orijentisan, jer je model domena kapsuliran.
- Neupućeni ponekad kao argument protiv ORM sistema iznose tvrnju da je ručno kodirana perzistencija brža. To je ekvivalentno tvrdnji da je asemblerski kod uvek brži od Java koda, ali koliko truda treba uložiti da bi se pisao asembler? U to smislu, pitanje je koliko ima smisla kodirati nešto ručno ako je na raspolaganju automatski alat. Ovakve tvrdnje se mogu razmatrati samo u kontekstu vremena i budžeta potrebnog za neko rešenje, a ne samo brzine.
- Ne morate biti stručnjak za različite DBMS sisteme (MySQL, Oracle, SQLServer, IBM Informix, SQLite itd.): Hibernate podržava rad sa gotovo svim DBMS sistemima[1], a svi trikovi i specifičnosti za određeni sistem već su ugrađeni. Takođe, ako se desi da se iz nekog razloga promeni DBMS sa podacima iz aplikacije, dovoljno je promeniti samo odgovarajuće parametre u konfiguracionoj datoteci, obezbediti odgovarajući drajver i upotrebiti Hibernate za ponovno generisanje šeme baze podataka; sama aplikacija ostaje netaknuta.

[1]Spisak podržanih DMBS sistema može se videti na adresi https://community.jboss.org/wiki/SupportedDatabases2

Podešavanje Hibernate projekta

Rad sa API-jem JDBC svodi se na izvršavanje zadataka veoma niskog nivoa, a programere aplikacija više zanima sam problem koji zahteva pristup podacima. Ono što programeri zaista žele jeste da napišu kod koji će pronalaziti i čuvati *objekte* u bazi podataka. Za to služi Hibernate. Da bi se koristio, prvo ga je neophodno preuzeti sa adrese http://www.hibernate.org/ (ovde ćemo koristiti verziju 3.2) i uključiti odgovarajuće jar arhive u biblioteku Java projekta. Takođe, potrebno je uključiti JDBC drajver za bazu podataka sa kojom će se raditi.

Hibernate aplikacije definišu perzistentne klase koje se preslikavaju u tabele baze podataka. Klase se definišu na osnovu analize problema koji se rešava, tj. predstavljaju model domena. Započećemo sa jednom perzistentnom klasom i objasniti kako ona izgleda, kako se za nju piše preslikavanje i šta može da se radi sa njenim instancama. Cilj sledećeg primera jeste da se poruke sačuvaju u bazi podataka i iz nje iščitaju radi prikaza. Uvešćemo zato prvu perzistentnu klasu `Poruka` koja predstavlja poruke koje se mogu prikazati:

```
package hello;
public class Poruka {
    private Long id;
    private String tekst;
    private Poruka sledecaPoruka;
    Poruka() {}
    public Poruka(String tekst) {
        this.tekst = tekst;
    }
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    public String getTekst() {
        return tekst;
    }
    public void setTekst(String tekst) {
        this.tekst = tekst;
    }
    public Poruka getSledecaPoruka() {
        return sledecaPoruka;
```

```
    }
    public void setSledecaPoruka(Poruka sledecaPoruka) {
        this.sledecaPoruka = sledecaPoruka;
    }
}
```

Klasa `Poruka` ima tri podatka člana: identifikator (id), tekst poruke i referencu na drugi objekat tipa `Poruka`. Identifikator omogućuje aplikaciji da pristupi primarnom ključu perzistentnog objekta u bazi podataka. Ako dve instance klase `Poruka` imaju istu vrednost identifikatora, to znači da predstavljaju isti zapis u bazi podataka. U ovom primeru kao tip identifikatora koristi se `Long`, ali to ne mora da bude tako. Hibernate dozvoljava gotovo svaki tip kao tip identifikatora. Primetićete da svi podaci članovi klase `Poruka` koriste stil JavaBean i odgovarajuće get/set metode. Klasa takođe ima konstruktor bez argumenata. Perzistentne klase će gotovo uvek biti takve; argument bez konstruktora je neophodan jer Hibernate koriste refleksiju sa ovim tipom konstruktora za instanciranje objekata. Instancama klase `Poruka` može da upravlja (tj. da ih perzistira) Hibernate, ali i ne mora. Pošto objekat klase `Poruka` ne implementira nikakve interfejse niti nasleđuje klase specifične za Hibernate, može se koristiti kao i svaka druga obična Java klasa:

```
Poruka poruka = new Poruka("Hello World");
System.out.println(poruka.getTekst());
```

Ovaj kod radi ono što se i očekuje, tj. ispisuje `Hello World` u konzoli. Dakle, perzistentna klasa se može koristiti u bilo kom kontekstu, tj. ne zahteva nikakav kontejner.

Preslikavanje klase u šemu baze podataka

Da bi objektno-relaciono preslikavanje bilo moguće, Hibernate zahteva informacije o tome kako tačno klasa `Poruka` treba da se perzistira. Drugim rečima, Hibernate mora da zna kako instance klase treba da se čuvaju u bazi podataka i učitavaju iz nje. Ti metapodaci mogu da se upišu u XML dokument za preslikavanje, u kome se između ostalog definiše kako se podaci članovi klase `Poruka` preslikavaju u kolone tabele `PORUKA`.

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="hello.Poruka" table="PORUKA">
    <id name="id" column="PORUKA_ID">
        <generator class="increment"/>
    </id>
```

```
        <property name="tekst" column="PORUKA_TEKST"/>
        <many-to-one name="sledecaPoruka" cascade="all"
            column="SLEDECA_PORUKA_ID" foreign-key="FK_SLEDECA_PORUKA"/>
        </class>
    </hibernate-mapping>
```

Dokument za preslikavanje kaže da klasa `Poruka` treba da se perzistira u tabelu `PORUKA`, da se podatak član preslikava u kolonu `PORUKA_ID`, da se podatak član `tekst` preslikava u kolonu `PORUKA_TEKST`, a da je podatak član `sledecaPoruka` u vezi tipa many-to-one koja se preslikava u kolonu sa spoljnim ključem (engl. *foreign key*) nazvanom `SLEDECA_PORUKA_ID`. Hibernate takođe generiše šemu baze podataka umesto nas i dodaje spoljni ključ nazvan `FK_SLEDECA_PORUKA` u bazi podataka. (Zasad zanemarite ostale detalje.) XML dokument nije težak za razumevanje, a možete ga lako pisati i održavati ručno.

Hibernate sada ima dovoljno informacija za generisanje svih SQL naredbi za umetanje, ažuriranje, brisanje i pronalaženje instanci klase `Poruka`. Te naredbe više ne morate da pišete ručno. Napravite datoteku Poruka.hbm.xml sa sadržajem iz prethodnog listinga i snimite je uz datoteku `Poruka.java` u paketu `hello`. Nastavak hbm je dogovor imenovanja koji se koristi u Hibernate zajednici, a većina programera obično čuva datoteke sa preslikavanjem uz izvorni kod odgovarajućih klasa.

Platforma Java EE uvela je pojednostavljen model u kome su XML konfiguracioni fajlovi opcioni, tj. preporučuje se korišćenje anotacija koje se unose direktno u izvorni kod, na mestu čiju konfiguraciju opisuju. Tako se i Hibernate XML preslikavanje sada može pisati u obliku anotacija (i taj način opisa metapodataka će povremeno biti korišćen u tekstu). Anotacije koje se koriste su zapravo JPA specifikacija i nalaze se u paketu `javax.persistence`; prethodno preslikavanje iz XML fajla sada izgleda ovako:

```
package hello;
import javax.persistence.*;

@Entity
@Table(table="PORUKA")
public class Poruka {

    @Id @GeneratedValue
    @Column("PORUKA_ID")
    private Long id;

    private String tekst;

    @ManyToOne(cascade=CascadeType.ALL)
```

```
    @JoinColumn(name="SLEDECA_PORUKA_ID")
    private Poruka sledecaPoruka;

    Poruka() {}
    public Poruka(String tekst) {
        this.tekst = tekst;
    }
    public Long getId() {
        return id;
    }
    private void setId(Long id) {
        this.id = id;
    }
    public String getTekst() {
        return tekst;
    }
    public void setTekst(String tekst) {
        this.tekst = tekst;
    }
    public Poruka getSledecaPoruka() {
        return sledecaPoruka;
    }
    public void setSledecaPoruka(Poruka sledecaPoruka) {
        this.sledecaPoruka = sledecaPoruka;
    }
}
```

JPA anotacija preslikava klasu označenu sa `@Entity` u tabelu baze `@Table`. Takođe, JPA prepoznaje anotaciju `@Id` koja se nalazi na polju (ne na metodi `getId()`) i na osnovu toga zaključuje da se svojstvima objekta tokom izvršavanja pristupa direktno (a ne preko metode `getId()`). Kada se anotacija `@Id` postavi na metod `getId()`, onda se svojstvima podrazumevano pristupa preko `get` i `set` metoda. Druge anotacije se zatim stavljaju na polja ili metode u skladu sa usvojenom strategijom.

Anotacije `@Table`, `@Column` i `@JoinColumn` nisu neophodne. Sva svojstva entiteta se automatski smatraju perzistentnim, sa podrazumevanom strategijom i nazivima tabela/kolona. Ovde su dodati samo zbog preglednosti i upoređivanja sa XML fajlom. Hibernate anotacije su skup osnovnih anotacija koje implementiraju JPA standard, kao i dodatnih anotacija koje se koriste u naprednijim i ređim podešavanjima. Ako je uvezen samo paket `javax.persistence.*`, to

znači da se koriste anotacije unutar JPA specifikacije; ako je uvezen i paket `org.hibernate.*`, koriste se izvorne Hibernate funkcionalnosti.

U vezi XML preslikavanja i anotacija, treba imati u vidu da XML format podržava sva moguća Hibernate preslikavanja, pa ako nešto ne može da se preslika pomoću anotacija, sigurno može da se napiše u XML fajlu. S druge strane, anotacije se ne mogu nadjačati pomoću XML fajla; celo preslikavanje tada mora da se napiše u XML-u.

Podešavanje za Hibernate

Hibernate se inicijalizuje tako što se objekat `SessionFactory` napravi iz objekta `Configuration`. Objekat klase `Configuration` se može posmatrati kao predstava datoteke se podešavanjima za Hibernate
(fajla `hibernate.properties` ili `hibernate.cfg.xml`). Ako se radi sa XML-fajlovima, treba upotrebiti naredbu:

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

U slučaju anotacija, koristi se klasa `AnnotationConfiguration`:

```
SessionFactory sessionFactory = new AnnotationConfiguration().
    configure().buildSessionFactory();
```

Kada se pozove konstruktor `Configuration()`, Hibernate traži datoteku `hibernate.properties` (tj. `hibernate.cfg.xml`) u korenu putanje klasa; ako je nađe, svojstva za Hibernate se učitavaju i dodaju objektu `Configuration` (taj objekat se može posmatrati kao predstava Hibernate podešavanja), a ako je ne nađe baca izuzetak. Ako podešavanja u XML konfiguracionoj datoteci nisu duplikati ranije podešenih svojstava, XML podešavanja zamenjuju ranija. Važno: konfiguracioni fajl mora da se nađe u korenu putanje klasa, izvan bilo kog paketa.

Konfiguraciona datoteka za Hibernate (`hibernate.cfg.xml`) koja treba da se nađe izvan svih paketa u izvornom kodu izgleda približno ovako:

```
<!DOCTYPE hibernate-configuration SYSTEM
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <property name="hibernate.connection.driver_class">
            org.hsqldb.jdbcDriver

        </property>

        <property name="hibernate.connection.url">
            jdbc:hsqldb:hsql://localhost

        </property>

        <property name="hibernate.connection.username"> sa </property>
```

```
        <property name="hibernate.connection.password"> sa </property>
        <property name="show_sql">true</property>
        <property name="format_sql">true</property>
        <mapping resource="hello/Poruka.hbm.xml"/>
    </session-factory>
</hibernate-configuration>
```

Deklaraciju tipa dokumenta (DTD) XML koristi za proveru dokumenta poređenjem sa DTD-om Hibernate konfiguracije. Primetite da to nije isti DTD kao za Hibernate XML datoteke sa preslikavanjem. Na početku datoteke nalaze se podešavanja za povezivanje sa bazom podataka. Hibernate mora da zna koji JDBC drajver, URL adresu, korisničko ime i lozinku koristite za povezivanje sa bazom. Takođe je omogućen ispis SQL naredbi koje izvršava Hibernate u konzoli, uz lepo formatiranje. Poslednji red u podešavanjima imenuje Hibernate XML datoteku sa preslikavanjem, jer objekat `Configuration` mora da zna za sve XML datoteke sa preslikavanjem pre nego što budete u stanju da napravite objekat `SessionFactory`. U većini Hibernate aplikacija objekat `SessionFactory` treba da se instancira samo jednom, tokom pokretanja aplikacije. Tu instancu zatim treba koristiti za dobijanje objekta `Session`. Objekat `SessionFactory` je thread-safe i može se deliti, dok je `Session` jednonitni objekat.

Kao šablon za inicijalizaciju obično se implementira klasa `HibernateUtil`:

```
import org.hibernate.*;
import org.hibernate.cfg.*;
public class HibernateUtil {
    private static SessionFactory sessionFactory;
    static {
        try {
            sessionFactory=new Configuration().
                configure().buildSessionFactory();
        } catch (Throwable ex) {
            throw new ExceptionInInitializerError(ex);
        }
    }
    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
    public static void shutdown() {
        getSessionFactory().close();
    }
}
```

Hibernate se pokreće u statičkom bloku za inicijalizaciju; taj blok se izvršava samo jednom prilikom učitavanja klase. Prvi poziv objekta `HibernateUtil` u aplikaciji učitava klasu, pravi `SessionFactory` i istovremeno inicijalizuje statičku promenljivu. Ako se pojavi problem, bilo koji `Exception` ili `Error` se baca iz statičkog bloka (zato se hvata `Throwable`). Pakovanje izuzetka u `ExceptionInInitializerError` je obavezno za statičke inicijalizatore. Kad god u aplikaciji zatreba pristup Hibernate objektu `Session`, do njega se može stići naredbom
`HibernateUtil.getSessionFactory().openSession();`

Snimanje i učitavanje objekata

Kod koji snima novi objekat klase `Poruka` u bazu podataka izgleda ovako:

```
package hello;
import java.util.*;
import org.hibernate.*;
import persistence.*;
public class HelloWorld {
public static void main(String[] args) {
    // Prva jedinica rada
    Session sesija = HibernateUtil.getSessionFactory().openSession();
    Transaction transakcija = session.beginTransaction();
    Poruka poruka = new Poruka("Hello World");
    Long idPoruka = (Long) sesija.save(poruka);
    transakcija.commit();
    sesija.close();
    // Druga jedinica rada
    Session novaSesija = HibernateUtil.getSessionFactory().openSession();
    Transaction novaTransakcija = newSession.beginTransaction();
    List poruka = novaSesija.createQuery("from Poruka p
        order by p.tekst asc").list();
    System.out.println("pronadjeno poruka:  " + poruke.size());
    for (Poruka p :  poruke) {
        System.out.println(p.getTekst());
    }
    novaTransakcija.commit();
    novaSesija.close();
    HibernateUtil.shutdown();
    }
}
```

Klasa ima standardnu Javinu metodu `main()` koja izvršava dve odvojene jedinice rada. Prva snima nov objekat `Poruka` u bazu, a druga učitava sve objekte i ispisuje njihov tekst u konzoli. Da biste pristupili bazi podataka, potrebni su sledeći interfejsi:

- `Session`. Hibernate klasa `Session` je jednonitni objekat koji modeluje jedinicu rada sa bazom podataka. Interfejs `Session` sastoji se od reda SQL naredbi koji u određenom trenutku treba da se sinhronizuje sa bazom podataka i mape perzistiranih instanci koje sesija nadgleda.

- `Transaction`. Koristi se za programsko postavljanje granica transakcija, koje nije obavezno.

- `Query`. Upit nad bazom podataka može da se napiše u objektno orijentisanom jeziku HQL (Hibernate Query Language) ili u običnom SQL-u (čemu treba pribeći tek ako smo sigurni da ne postoji drugo rešenje). Interfejs `Query` omogućuje pisanje upita, povezivanje argumenata u pripremljenim upitima i izvršavanje upita na različite načine.

Kada se izvrši prva jedinica rada, izvršava se nešto slično SQL naredbi:

```
insert into PORUKA (PORUKA_ID, PORUKA_TEKST, SLEDECA_PORUKA_ID)

    values (1, 'Hello World', null)
```

Kolona `PORUKA_ID` se inicijalizuje vrednošću 1 koja nigde nije dodeljena, pa bi se očekivalo da bude NULL. Međutim, podatak član `id` je poseban jer je to identifikator koji sadrži generisanu jedinstvenu vrednost. Tu vrednost Hibernate dodeljuje instanci klase `Poruka` kada se pozove metoda `save()`.

U drugoj jedinici rada string "`from Poruka p order by p.tekst asc`" je Hibernate upit izražen jezikom HQL. Taj upit se interno prevodi u sledeći SQL kada se pozove metoda `list()`:

```
select p.PORUKA_ID, p.PORUKA_TEKST, p.SLEDECA_PORUKA_ID
    from PORUKA p order by p.PORUKA_TEKST asc
```

Ako izvršite ovaj metod `main()` , na konzoli će se ispisati:

```
pronadjeno poruka:  1 Hello World
```

Pre nego što objasnimo kako se podešava Hibernate, objasnimo još dva svojstva: automatsku proveru da li je objekat u sesiji izmenjen (engl. *dirty checking*) i kaskadiranje u trećoj jedinici rada koju ćemo dodati u `main()`:

```
//treca jedinica rada
Session trecaSesija = HibernateUtil.getSessionFactory().openSession();
Transaction trecaTransakcija = trecaSesija.beginTransaction();
// idPoruka sadrži id prve poruke
poruka = (Poruka) trecaSesija.get(Poruka.class, idPoruka);
poruka.setTekst("Dobrodosli");
```

```
poruka.setSledecaPoruka(new Poruka("Vodi me kuci"));
trecaTransakcija.commit();
trecaSesija.close();
```

Ovaj kod izvršava tri SQL naredbe u istoj transakciji baze podataka:

```
select p.PORUKA_ID, p.PORUKA_TEKST, p.SLEDECA_PORUKA_ID
    from PORUKA p where p.PORUKA_ID = 1
insert into PORUKA (PORUKA_ID, PORUKA_TEKST, SLEDECA_PORUKA_ID)
   values
(2, 'Vodi me kuci', null)
update PORUKA set PORUKA_TEKST = 'Dobrodosli',
    SLEDECA_PORUKA_ID = 2 where PORUKA_ID = 1
```

Primetite kako je Hibernate registrovao izmenu svojstava `tekst` i `sledecaPoruka` prve poruke i automatski je ažurirao bazu podataka, odnosno automatski je obavio proveru da li je objekat u sesiji izmenjen (engl. *dirty checking*). Ta funkcionalnost štedi napor da se eksplicitno zahteva da Hibernate ažurira bazu kad god se promeni stanje objekta u jedinici rada. Slično, nova poruka je perzistirana kada je referenca na nju kreirana iz prve poruke. Ta funkcionalnost se zove kaskadno čuvanje (engl. *cascade save*) i štedi napor eksplicitnog perzistiranja novog objekta pozivom metode `save()`, pod uslovom da se do novog objekta može stići preko objekta koji je već perzistiran. Primetite takođe da redosled SQL naredbi nije isti kao redosled kojim smo postavili vrednosti podataka članova objekta. Hibernate koristi napredni algoritam za određivanje efikasnog redosleda izvršavanja SQL naredbi da bi se izbegla ograničenja koja nameću spoljni ključevi; to svojstvo se zove *transactional write-behind*.

Ako sada pokrenemo aplikaciju, prikazao bi se sledeći rezultat:

```
pronadjeno poruka:  2 Dobrodosli Vodi me kuci
```

Kreiranje šeme baze podataka

Šemu baze podataka možete napraviti ručno, pisanjem DDL datoteke sa SQL naredbama CREATE, ili što je mnogo zgodnije, možete da prepustite da Hibernate obavi taj zadatak i napravi podrazumevanu šemu za aplikaciju. Da bi Hibernate automatski generisao SQL DDL, neophodno je da preslikavanje bude definisano. Alat koja se koristi za generisanje šeme je hbm2ddl, a odgovarajuća klasa `org.hibernate.tool.hbm2ddl.SchemaExport`.

Postoji više načina za pokretanje ovog alata, a ovde ćemo opisati dva: podešavanjem svojstva `hibernate.hbm2ddl.auto` u konfiguracionoj datoteci za Hibernate (`hibernate.cfg.xml`) ili iz programskog koda. Ako se ovo svojstvo podesi na create, naredbe DROP će pratiti naredbe CREATE prilikom kreiranja objekta `SessionFactory`. Sa podešavanjem create-drop, prilikom uništavanja objekta `SessionFactory` generišu se naredbe DROP, pa se nakon svakog izvršavanja dobija prazna šema baze podataka.

Izvoz šeme baze podataka iz programa je jednostavan:

```
Configuration cfg = new Configuration().configure();
SchemaExport schemaExport = new SchemaExport(cfg);
schemaExport.create(false, true);
```

Nov objekat `SchemaExport` se pravi iz objekta `Configuration`; sva podešavanja (drajver baze podataka, URL za povezivanje itd.) prosleđuju se konstruktoru klase `SchemaExport`. Metoda `create(false, true)` započinje generisanje DDL datoteke, bez prikaza SQL naredbi na standardnom izlazu (parametar false), ali uz trenutno izvršavanje u bazi podataka (parametar true).

Još jedna korisna vrednost svojstva `hibernate.hbm2ddl.auto` tokom razvoja je `validate`. Ono pokreće alat `SchemaValidator` koje tokom pokretanja aplikacije proverava da li se preslikavanje poklapa sa JDBC metapodacima, odnosno da li se stanje u bazi i preslikavanje poklapaju. Provera se može pozvati i iz programa, pozivom klase `SchemaValidator`:

```
Configuration cfg = new Configuration().configure();
new SchemaValidator(cfg).validate();
```

Ako Hibernate ustanovi nepoklapanje između šeme baze i preslikavanja, baca izuzetak.

Preslikavanje: koncepti i strategije

Preslikavanje identifikatora

Pre nego što pređemo na objašnjenje kako Hibernate upravlja identitetima u bazi podataka, neophodno je da se podsetimo razlike između identičnosti i jednakosti objekata. Identičnost objekata, koja se u Javi proverava operatorom ==, znači da reference na objekte pokazuju na istu lokaciju u memoriji virtuelne mašine. Jednakost (ekvivalencija) objekata proverava se metodom `equals()` koja vraća true ako dva različita (neidentična) objekta sadrže istu vrednost.

Persistencija dodatno usložnjava stvar. Perzistentni objekat je predstava određenog zapisa iz tabele baze podataka u memoriji. Zajedno sa Javinim identitetom (memorijskom lokacijom) i ekvivalentnošću objekata, proverava se i identitet iz baze podataka. Dakle, postoje tri načina za prepoznavanje objekata:

- Objekti su identični ako u Javinoj virtuelnoj mašini zauzimaju istu memorijsku lokaciju. Identitet se može proveriti pomoću operatora == .
- Objekti su ekvivalentni ako imaju istu vrednost, kao što je definisano u metodi `equals(Object o)`. Klase koje eksplicitno ne redefinišu ovaj metod nasleđuju implementaciju tog metoda iz klase `java.lang.Object`.
- Objekti koji se čuvaju u relacionoj bazi podataka su identični ako predstavljaju isti zapis, ili što je isto, ako imaju istu tabelu i primarni ključ. Ovaj pojam je poznat kao identitet baze podataka. Osim operacija za testiranje jednakosti objekata (`a == b`) i njihove ekvivalencije (`a.equals(b)`), može se koristiti i `a.getId().equals(b.getId())` za testiranje identičnosti u bazi podataka.

Svojstvo identifikatora objekta (identifier) je posebno, i njegova vrednost je primarni ključ zapisa baze podataka koju predstavlja perzistentni objekat. U našem primeru označavali smo ga kao `id`. Običan (prost) identifikator se pomoću anotacija preslikava na sledeći način:

```
@Id
@Column (name="CATEGORY_ID")
```

Identifikator je preslikan u kolonu primarnog ključa `CATEGORY_ID`. Hibernate tip za ovo svojstvo je `long`, što se u većini baza podataka preslikava u tip BIGINT. Dobra je Hibernate praksa da se u novim aplikacijama umesto "prirodnih" primarnih ključeva (tj. kolona tabele koje jedinstveno određuju zapis) koriste "veštački" generisani primarni ključevi. Postoji više pristupa generisanju veštačkih primarnih ključeva, ali se najčešće koriste parametri `native` i `increment`. Podešavanje `native` je uobičajen izbor za nove aplikacije; kada je reč o nasleđenom kodu, ponekad se moraju koristiti prirodni, čak i složeni primarni ključevi, ali se time ovde nećemo baviti.

Preslikavanje klasa

Hibernate koristi refleksiju da bi odredio Javin tip podatka člana klase. To znači da su sledeća preslikavanja ekvivalentna:

```
<property name="opis" column="OPIS" type="string"/>
<property name="opis" column="OPIS"/>
```

Moguće je izostaviti čak i ime kolone ako je isto kao ime podatka člana (uz zanemarivanje velikih i malih slova). Korisno je (ali ne i obavezno) navesti atribut not-null, pošto u tom slučaju Hibernate može da prijavi nedozvoljene null vrednosti bez provere u bazi podataka:

```
<property name="pocetnaCena" column="POCETNA_CENA" not-null="true"/>
```

Ako se koriste anotacije, nije potrebno navoditi tip:

```
@Column("POCETNA_CENA", nullable=false)
String pocetnaCena;
```

Preslikavanje nasleđivanja

Jednostavna strategija za preslikavanje klasa u tabele baze podataka mogla bi da bude "jedna tabela za svaku perzistentnu klasu". Taj pristup izgleda jednostavno i zaista radi dobro sve dok se ne naiđe na nasleđivanje. Nasleđivanje je odličan primer na kome se uočava neslaganje između objektno orijentisanog i relacionog sveta. Postoje različiti pristupi preslikavanja hijerarhije nasleđivanja:

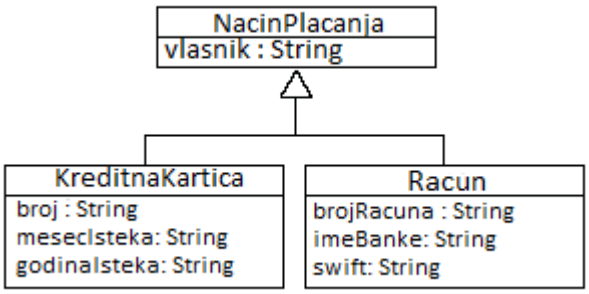
1. Po jedna tabela za svaku konkretnu klasu sa implicitnim polimorfizmom. Izbegava direktno preslikavanje nasleđivanja i umesto toga tokom izvršavanja koristi polimorfno ponašanje klasa.

2. Po jedna tabela za svaku konkretnu klasu. Odbacuje polimorfizam i nasleđivanje i nasleđuje relacije direktno iz SQL šeme.

3. Jedinstvena tabela za celu hijerarhiju klasa. Omogućuje polimorfizam denormalizacijom SQL šeme i koristi polje sa diskriminatorom koje sadrži informaciju o tipu.

4. Posebna tabela za svaku izvedenu klasu. Predstavlja relacije nasleđivanja kao relacije spoljašnjeg ključa.

U nastavku ćemo opisati pristupe 3 i 4. Pretpostavimo da imamo hijerarhiju nasleđivanja kao na sledećoj slici:



Jedinstvena tabela za celu hijerarhiju klasa Celokupna hijerarhija klasa preslikava se u jednu tabelu koja sadrži polja za sve podatke članove svih klasa koje čine hijerarhiju. Konkretna izvedena klasa koju predstavlja određeni zapis prepoznaje se na osnovu vrednosti tipa u diskriminatorskom polju. Ovaj način preslikavanja se pokazao kao najjednostavniji i najbolji u pogledu performansi.

Treba obratiti pažnju na dva detalja: prvo, polja za podatke članove deklarisane u izvedenim klasama moraju da dozvole null vrednosti (tj. ne smeju da se deklarišu kao not-null). Kada bi se u izvedenim klasama dozvolila nenull svojstva, gubitak NOT NULL ograničenja mogao bi da predstavlja ozbiljan problem po pitanju integriteta podataka. Drugo, pošto se na ovaj način prave funkcionalne zavisnosti između polja koja ne uzimaju vrednosti null, narušava se normalizacija baze. Za preslikavanje nasleđivanja ovim pristupom koristi se sledeće anotacije:

```
@Entity
@Table("NACIN_PLACANJA")
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="NACIN_PLACANJA_TIP", discriminatorType
= DiscriminatorType.STRING)
public abstract class NacinPlacanja {
```

```
    @Id @GeneratedValue
    @Column(name="NACIN_PLACANJA_ID")
    private Long id;

    @Column(name="VLASNIK")
    private String vlasnik;
...

@Entity
@Table("KREDITNA_KARTICA")
@DiscriminatorValue("KK")
public class KreditnaKartica extends NacinPlacanja {
    @Column("KK_BROJ")
    private String broj;

    @Column(KK_MESEC_ISTEKA)
    private Integer mesecIsteka;
    ...

@Entity
@Table("RACUN")
@DiscriminatorValue("RN")
public class Racun extends NacinPlacanja {

    @Column("BR_RACUNA")
    private String brojRacuna;
    ...
```

Da bi se perzistentne klase u hijerarhiji razlikovale, neophodno je dodati posebno polje, tzv. diskriminator. To nije svojstvo perzistentne klase, već ga interno koristi Hibernate. U našem slučaju, ime polja je `NACIN_PLACANJA_TIP` i može da ima string vrednosti "KK" ili "RN". Hibernate automatski upisuje i čita vrednosti diskriminatora. Podaci članovi izvedene klase su preslikani u polja tabele `NACIN_PLACANJA`. Ne zaboravite da ograničenja NOT NULL za podatke članove izvedenih klasa nisu dozvoljena jer instanca klase `Racun` nece imati podatak član `mesecIsteka`, pa polje `KK_MESEC_ISTEKA` za odgovarajući zapis mora imati vrednost NULL.

Posebna tabela za svaku izvedenu klasu Svaka klasa i izvedena klasa koja deklariše perzistentna svojstva, uključujući i apstraktne klase, ima sopstvenu tabelu. Tabela sadrži samo polja za podatke članove koji se ne nasleđuju (odnosno za podatke članove koji su deklarisani u samoj izvedenoj klasi), zajedno sa primarnim ključem koji je istovremeno spoljni ključ tabele roditeljske klase. U primeru sa slike, ako je neka instanca izvedene klase `KreditnaKartica` perzistirana, vrednosti podataka članova koje nasleđuje iz klase `NacinPlacanja` smeštaju se kao nov zapis u tabeli `NACIN_PLACANJA`. Novi zapis koji se dodaje u tabelu `KREDITNA_KARTICA` sadrži samo podatke članove iz te klase. Dva nova zapisa se povezuju pomoću deljene vrednosti primarnog ključa. Kasnije se instanca izvedene klase može dobiti iz baze podataka ukrštanjem tabele izvedene klase sa tabelom osnovne klase. Osnovna prednost ovakvog pristupa je u tome što je SQL šema normalizovana, i što se lako razvija dodavanjem novih izvedenih klasa. Nasleđivanje se postiže elementom <joined-subclass>, odnosno sledećom anotacijom:

```
@Entity
@Table("NACIN_PLACANJA")
@Inheritance(strategy=InheritanceType.JOINED)
public abstract class NacinPlacanja {
    @Id @GeneratedValue
    @Column(name="NACIN_PLACANJA_ID")
    private Long id;

    @Column(name="VLASNIK")
        private String vlasnik;
    ...
```

U izvedenim klasama ne morate da zadajete kolonu po kojoj se spajaju tabele ako kolona sa primarnim ključem tabele potklase ima isto ime kao kolona sa primarnim ključem u roditeljskoj klasi; ako imaju različite nazive, onda se kolona primarnog ključa u izvedenoj klasi označava anotacijom `@PrimaryKeyJoinColumn`.

```
@Entity
@Table("KREDITNA_KARTICA")
@PrimaryKeyJoinColumn(name="KREDITNA_KARTICA_ID")
public class KreditnaKartica extends NacinPlacanja {
        @Column("KK_BROJ")
        private String broj;

        @Column(KK_MESEC_ISTEKA)
        private Integer mesecIsteka;
        ...
```

```
@Entity
@Table(name="RACUN")
@PrimaryKeyJoinColumn("RACUN_ID")
public class Racun extends NacinPlacanja {

    @Column("BR_RACUNA")
    private String brojRacuna;
    ...
```

Primetite da je za tabelu `KREDITNA_KARTICA` potreban primarni ključ, koji istovremeno služi kao spoljni ključ za tabelu `NACIN_PLACANJA`. Pronalaženje objekta `KreditnaKartica` zahteva ukrštanje obeju tabela.

Hibernate tipovi

Ugrađeni Hibernate tipovi za preslikavanje obično imaju isto ime kao Javin tip koji preslikavaju. Međutim, ponekad postoji više Hibernate tipova za isti Javin tip. Moguće je čak i definisati sopstvene tipove za preslikavanje. O SQL tipovima obično ne morate da brinete dok koristite Hibernate za pristup podacima i definiciju SQL šeme. Na primer, sistem tipova je dovoljno pametan da zna da izabere pravi SQL tip u zavisnosti od dužine podatka. Najočigledniji slučaj je `String`: ako zadate atribut `length`, Hibernate će izabrati odgovarajući SQL tip za dijalekt (DBMS) sa kojim radite.

Kada je reč o predstavljanju datuma, možete koristiti `java.util.Date`, `java.util.Calendar` ili potklase datuma iz paketa `java.sql` (što nije preporučljivo), ali koji kod da izaberete, budite dosledni u kodu. Ako svojstvo tipa `java.util.Date` preslikate tipom `timestamp`, Hibernate će iz baze vratiti tip `java.sql.Timestamp`, što može da prouzrokuje probleme prilikom poređenja datuma. Zapravo, JPA anotacije zahtevaju da naznačite da li se radi o datumu ili vremenu pomoću anotacije `@Temporal`:

```
@Temporal(TemporalType.TIMESTAMP)
@Column(nullable=false)
private Date startDate;
```

Kada je reč o preslikavanju binarnih vrednosti, JPA koristi anotaciju `@Lob`, a Hibernate ih odmah učitava. Za svojstva tipa `Serializable`, vrednost se konvertuje u tok bajtova koji se čuvaju u koloni tipa VARBINARY (ili nešto slično); kada se učita objekat čije je to svojstvo, ono se deserijalizuje. Ovo treba koristiti uz veliki oprez i samo za privremene podatke (podešavanja korisnika, podatke o prijavljivanju i sl.)

Tip preslikavanja	Javin tip	Standardni SQL tip
date	java.util.Date or java.sql.Date	DATE
time	java.util.Date or java.sql.Time	TIME
timestamp	java.util.Date or java.sql.Timestamp	TIMESTAMP
calendar	java.util.Calendar	TIMESTAMP
calendar_date	java.util.Calendar	DATE
binary	byte[]	VARBINARY
text	java.lang.String	CLOB
clob	java.sql.Clob	CLOB
blob	java.sql.Blob	BLOB
serializable	Bilo koja Java klasa koja implementira java.io.Serializable	VARBINARY

Preslikavanje kolekcija

Većina Hibernate početnika se sa preslikavanjem kolekcija i različitih asocijacija sretne kada prvi put pokuša da preslika tipičan odnos roditelj-dete. Međutim, da bi se to preslikavanje razumelo i pravilno uradilo, neophodno je razumeti osnovne pojmove. Java nudi bogat API za kolekcije, pa se uvek može pronaći implementacija koja najbolje odgovara problemu koji se modelira. U nastavku ćemo objasniti kako se preslikavaju najčešće korišćene kolekcije: liste, skupovi i mape.

Hibernate zna kako da preslika kolekcije i za to koristi sledeće elemente:

- Skup (`java.util.Set`) se preslikava elementom <set>. Kolekcija se inicijalizuje tipom `java.util.HashSet`. Redosled dodavanja elemenata u skup se ne čuva, a duplikati nisu dozvoljeni. Ovo je kolekcija koja se najčešće koristi u Hibernate aplikacijama.
- Sortiran skup (`java.util.SortedSet`) perslikava se elementom <set> u kome se atribut sortiranja zadaje komparatorom ili prirodnim redosledom, u zavisnosti od tipa elemenata u skupu. Kolekcija se inicijalizuje tipom `java.util.TreeSet`.
- Lista (`java.util.List`) se preslikava elementom <list>, uz čuvanje pozicije svakog elementa u dodatnoj koloni index u tabeli kolekcije. Lista se inicijalizuje tipom `java.util.ArrayList`.
- Mapa (`java.util.Map`) se preslikava elementom <map>, uz čuvanje parova ključ-vrednost. Za inicijalizaciju se koristi tip `java.util.HashMap`.

- Sortirana mapa (`java.util.SortedMap`) se preslikava elementom <map>, pri čemu se atribut sortiranja zadaje komparatorom ili prirodnim redosledom, u zavisnosti od tipa mape. Kolekcija se inicijalizuje tipom `java.util.TreeMap`.

Preslikavanje skupova Posmatrajmo primer implementacije skupa imena datoteka tipa `String`, kao na sledećoj slici:

PROIZVOD

PROIZVOD_ID	NAZIV
1	Krevet
2	Sto
3	Lampa

PROIZVOD_SLIKA

PROIZVOD_ID	FILENAME
1	krevet1.jpg
1	krevet2.jpg
2	sto1.jpg

Prvo treba dodati svojstvo kolekcije u klasu `Proizvod`:

```
private Set<String> slike = new HashSet<String>();
...
public Set<String> getSlike() {
    return this.slike;
}
public void setSlike(Set<String> slike) {
    this.slike = slike;
}
```

Zatim treba dodati preslikavanje u XML datoteku klase `Proizvod`:

```
<set name="slike" table="PROIZVOD_SLIKA">
    <key column="PROIZVOD_ID"/>
    <element type="string" column="FILENAME" not-null="true"/>
</set>
```

Imena datoteka čuvaju se u tabeli kolekcije `PROIZVOD_SLIKA`. Sa tačke gledišta baze podataka, ta tabela je poseban entitet, ali Hibernate to od nas krije. Element <key> deklariše polje koje služi kao spoljni ključ u tabeli kolekcije i referencira primarni ključ `PROIZVOD_ID` entiteta u kome se nalazi skup. Pošto skup ne može da sadrži duplikate, primarni ključ kolekcije `PROIZVOD_SLIKA` je složen i sastoji se od oba polja u deklaraciji <set>: `PROIZVOD_ID` i `FILENAME`.

Preslikavanje liste

Prvo je potrebno promeniti klasu `Proizvod`:

```
private List<String> slike = new ArrayList<String>();
...
public List<String> getSlike() {
    return this.slike;
}
public void setSlike(List<String> slike) {
    this.slike = slike;
}
```

Preslikavanje <list> zahteva dodavanje polja index u tabelu kolekcije. Polje index definiše položaj elementa u kolekciji. Na taj način Hibernate je u stanju da očuva redosled elemenata u kolekciji. Preslikavanje je sledeće:

```
<list name="slike" table="PROIZVOD_SLIKA">
    <key column="PROIZVOD_ID"/>
    <list-index column="POZICIJA"/>
    <element type="string" column="FILENAME" not-null="true"/>
</list>
```

Primarni ključ tabele kolekcije je složen od polja `PROIZVOD_ID` i `POZICIJA`. Duplikati elemenata (`FILENAME`) nisu dozvoljeni, što je u skladu sa semantikom liste. Indeks perzistentne liste počinje od nule (to se može promeniti elementom <list-index base="1".../>). Hibernate dodaje null elemente u Java listu ako brojevi indeksa u bazi podataka nisu kontinualni.

PROIZVOD

PROIZVOD_ID	NAZIV
1	Krevet
2	Sto
3	Lampa

PROIZVOD_SLIKA

PROIZVOD_ID	POZICIJA	FILENAME
1	0	krevet1.jpg
1	1	krevet2.jpg
1	2	krevet3.jpg

Preslikavanje mape

Pretpostavimo sada da slike za proizvod osim imena datoteke imaju i nazive koje su im dodelili korisnici. Jedan od načina da se to modelira u Javi je mapa, pri čemu bi korisnički nazivi bili ključevi, a imena datoteka vrednosti mape. Java klasa se menja:

```
private Map<String, String> slike = new HashMap<String, String>();
...
public Map<String, String> getSlike() {
    return this.slike;
}
public void setSlike(Map<String, String> slike) {
    this.slike = slike;
}
```

Preslikavanje mape je slično preslikavanju liste:

```
<map name="slike" table="PROIZVOD_SLIKA">
    <key column="PROIZVOD_ID"/>
    <map-key column="IME_SLIKE" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

Primarni ključ tabele kolekcije je složen od `PROIZVOD_ID` i `IME_SLIKE`. Polje `IME_SLIKE` sadrži ključeve mape. Ni tu nisu dozvoljeni duplikati.

PROIZVOD

PROIZVOD_ID	NAZIV
1	Krevet
2	Sto
3	Lampa

PROIZVOD_SLIKA

PROIZVOD_ID	IME_SLIKE	FILENAME
1	slika jedan	krevet1.jpg
1	slika dva	krevet2.jpg
1	slika tri	krevet3.jpg

Sortirane i uređene kolekcije

Kada je reč o perzistentnim kolekcijama, reči sortiran (engl. *sorted*) i uređen (engl. *ordered*) nemaju isto značenje. Sortirana kolekcija je sortirana u memoriji pomoću Javinog komparatora. Uređena kolekcija je uređena na nivou baze podataka pomoću SQL upita sa klauzulom `order by`. Ako želimo sortiranu mapu, na primer, prvo ćemo kao tip interfejsa `Map` u klasi zadati `TreeMap`, a zatim ćemo dodati atribut `sort="natural"`:

```
<map name="slike" table="PROIZVOD_SLIKA" sort="natural">
    <key column="PROIZVOD_ID"/>
    <map-key column="IME_SLIKE" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

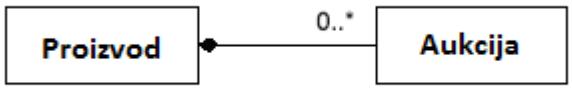
Ako želimo uređenu mapu, možemo da sortiramo elemente u bazi pomoću sledećeg preslikavanja:

```
<map name="slike" table="PROIZVOD_SLIKA" order-by="IME_SLIKE asc">
    <key column="PROIZVOD_ID"/>
    <map-key column="IME_SLIKE" type="string"/>
    <element type="string" column="FILENAME" not-null="true"/>
</map>
```

Isto se može uraditi i za skupove.

Preslikavanje kardinalnosti

Upravljanje vezama između klasa i relacijama između tabela je srce ORM sistema. Pretpostavimo da želimo da modeliramo sledeću vezu:



U opisivanju i klasifikaciji asocijacija koristimo termin kardinalnost (engl. *multiplicity*). U našem primeru, kardinalnost određuju dve informacije:

- Može li biti više od jedne Aukcije za određen Proizvod?
- Može li biti više Proizvoda za određenu Aukciju?

Iz modela sa slike zaključujemo da je veza od Aukcije ka Proizvodu tipa *many-to-one*, a obrnuta veza od Proizvoda ka Aukciji tipa *one-to-many*. Postoje još dve mogućnosti: *many-to-many* i *one-to-one*, kojima se ovde nećemo baviti jer se ređe sreću.

Preslikavanje many-to-one

Java implementacija klase Aukcija izgleda ovako:

```
public class Aukcija {
    ...
    private Proizvod proizvod;
    public void setProizvod(Proizvod proizvod) {
        this.proizvod = proizvod;
    }
    public Proizvod getProizvod() {
        return proizvod;
```

```
    }
    ...
}
```

Hibernate preslikavanje za ovu vezu izgleda ovako:

```
<class name="Aukcija" table="AUKCIJA">
...
<many-to-one name="proizvod" column="PROIZVOD_ID"
    class="Proizvod" not-null="true"/>
</class>
```

Polje `PROIZVOD_ID` u tabeli `AUKCIJA` je spoljni ključ ka primarnom ključu tabele `PROIZVOD`. Klasa `Proizvod` koja je cilj veze navedena je eksplicitno, ali to nije neophodno jer Hibernate može da odredi tu klasu pomoću refleksije. Atribut not-null je dodat zato što ne može da postoji aukcija bez proizvoda (spoljni ključ `PROIZVOD_ID` u tabeli `AUKCIJA` ne može da bude NULL).

Pomoću JPA, ovo preslikavanje koristi anotaciju `@ManyToOne`, na polju ili na get metodi, u zavisnosti od toga koju strategiju za pristup entitetu izaberete (a što je određeno pozicijom anotacije `@Id`):

```
public class Aukcija {

    @ManyToOne
    @JoinColumn(name="PROIZVOD_ID", nullable = false)
    private Proizvod proizvod;
    ...
}
```

Važno je ovde razumeti da biste kompletnu aplikaciju mogli da napišete samo pomoću ovog preslikavanja. Ne morate obavezno da preslikate drugu stranu ove veze između klasa, jer ste već preslikali sve što se nalazi u SQL šemi (kolonu sa spoljnim ključem). Ako vam je potrebna instanca `Proizvoda` za koju postoji određena `Aukcija`, treba samo da pozovete `aukcija.getProizvod()`. Ako vam trebaju sve aukcije za određeni proizvod, možete da napišete upit. S druge strane, jedan od razloga za korišćenje ORM alata kao što je Hibernate je baš to što *ne želite* da pišete taj upit.

Kako ćemo vezu između `Aukcija` i `Proizvoda` učiniti dvosmernom? Želimo da budemo u stanju da lako stignemo do svih aukcija za određeni proizvod bez eksplicitnog upita, tako što ćemo se kretati kroz mrežu perzistentnih objekata. Najlakše je to postići tako što ćemo u klasu `Proizvod` dodati kolekciju `Aukcija`:

```
public class Proizvod {
    ...
    private Set<Aukcija> aukcije = new HashSet<Aukcija>();
```

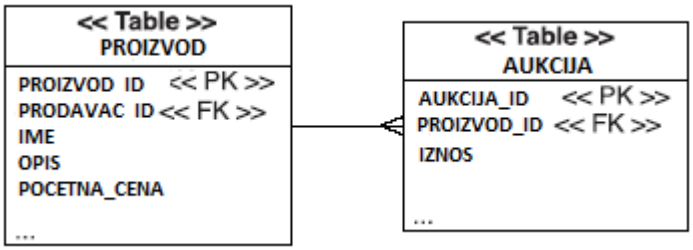
```
    public void setAukcije(Set<Aukcija> aukcije) {
        this.aukcije = aukcije;
    }
    public Set<Aukcija> getAukcije() {
        return aukcije;
    }
    public void dodajAukciju(Aukcija aukcija) {
        aukcija.setProizvod(this);
        aukcije.add(aukcija);
    }
       ...
}
```

Na preslikavanje kolekcija uvek gledajte kao na dodatnu pogodnost, a ne nezaobilaznu stavku; ako vam se učini da je previše teško, nemojte to uraditi.

Osnovno preslikavanje za ovu vezu tipa one-to-many izgleda ovako:

```
<class name="Proizvod" table="PROIZVOD">
    ...
        <set name="aukcije"> <key column="PROIZVOD_ID"/>
        <one-to-many class="Aukcija"/>
        </set>
</class>
```

Ako ovo preslikavanje uporedimo sa preslikavanjem skupa opisanom u ranijem odeljku, primećujemo da se sadržaj kolekcije preslikava drugačijim elementom, <one-to-many>. To označava da kolekcija sadrži reference na instance nekog preslikanog entiteta. Preslikano polje definisano elementom <key> je spoljni ključ, tj. polje `PROIZVOD_ID` tabele `AUKCIJA`, što je isto polje koje je već preslikano sa druge strane veze. Izgled tabela u bazi se nije promenio nakon dodavanja ovog preslikavanja i izgeda ovako:



Jedina razlika je što sada nedostaje atribut not null="true"; sada imamo dve različite jednosmerne veze preslikane u isto polje koje sadrži spoljni ključ. Koja strana veze upravlja tim poljem? Tokom izvršavanja, ista vrednost spoljašnjeg ključa u memoriji je predstavljena sa dve vrednosti: svojstvom `proizvod` klase `Aukcija` i elementom kolekcije `aukcije` koja se nalazi u objektu `Proizvod`. Pretpostavimo da aplikacija izmeni vezu, npr. dodavanjem aukcije proizvodu:

```
aukcija.setProizvod(proizvod);
aukcije.add(aukcija);
```

U ovoj situaciji Hibernate prepoznaje dve izmene u različitim instancama objekata koje treba perzistirati, a nalaze se u memoriji. Sa tačke gledišta baze podataka, obe promene treba da se svedu na izmenu spoljnog ključa, tj. kolone `PROIZVOD_ID` u tabeli `AUKCIJA`. Hibernate nije u stanju da neprimetno registruje da se te dve izmene odnose na isto polje u bazi, jer mu ništa u ovom trenutku ne govori da se radi o dvosmernoj vezi. Drugim rečima, isto polje je dvaput preslikano, bez obzira što je to urađeno u dva različita fajla. Da bi se veza učinila dvosmernom, potrebno je dodati atribut `inverse` kojim se označava da je kolekcija slika u ogledalu veze <many-to-one> sa druge strane:

```
<class name="Proizvod" table="PROIZVOD">
    ...
    <set name="aukcije" inverse="true">
        <key column="PROIZVOD_ID"/>
        <one-to-many class="Aukcija"/>
    </set>
</class>
```

Bez atributa `inverse` Hibernate pokušava da izvrši dve SQL naredbe koje menjaju isti spoljni ključ. Zadavanjem `inverse="true"` eksplicitno se kaže koju stranu veze ne treba da sinhronizuje sa bazom podataka. U ovom primeru, Hibernate će znati da u bazu podataka treba da pošalje samo izmene koje su napravljene na strani veze kod `Aukcije`, i da zanemari promene u kolekciji aukcije. Ako se u aplikaciji pozove `proizvod.getAukcije().add(aukcija)`, to se neće odraziti na bazu podataka! Ono što želimo da postignemo treba da se uradi na drugoj strani: `aukcija.setProizvod(proizvod)`.

Ako se koriste JPA anotacije, ekvivalent atributa `inverse` je `mappedBy`, koji mora da imenuje inverzno svojstvo entiteta na koji se odnosi:

```
public class Proizvod {
...
@OneToMany(mappedBy="proizvod")
private Set<Aukcija> aukcije = new HashSet<Aukcija>();
...
}
```

Na opisan način napravili smo funkcionalnu dvosmernu vezu many-to-one. Ako želimo da napravimo odnos roditelj-dete (engl. *parent-child relationship*), nedostaje nam još nešto.

Kaskadiranje stanja objekata

Pojam roditelj-deca podrazumeva da roditeljski entitet vodi računa o svojoj deci. Praktično, kada se pravilno implementira, upravljanje odnosom roditelj-deca trebalo bi da zahteva manje koda jer veći deo posla treba da se obavlja automatski. Sledeći kod kreira nov `Proizvod` (roditelj) i novu instancu `Aukcija` (dete):

```
Proizvod novProizvod = new Proizvod();
Aukcija novaAukcija = new Aukcija();
novProizvod.addAukcija(novaAukcija);
session.save(novProizvod);
session.save(novaAukcija);
```

Drugi poziv metode `session.save()` bi bio suvišan u slučaju prave veze roditelj-dete. Kada se instancira nov objekat `Aukcija` i doda se `Proizvodu`, aukcija bi trebalo automatski da se perzistira, bez dodatne operacije `save()`; to se zove tranzitivna perzistencija (engl. *transitive persistence*). Da bi se to omogućilo, u XML preslikavanje dodaje se opcija `cascade`:

```
<class name="Proizvod" table="PROIZVOD">
    ...
    <set name="aukcije" inverse="true" cascade="save-update">
        <key column="PROIZVOD_ID"/>
        <one-to-many class="Aukcija"/>
    </set>
</class>
```

Atribut `cascade="save-update"` omogućuje tranzitivnu perzistenciju za instance `Aukcija`, ako je određen objekat `Aukcija` referenciran u kolekciji perzistiranog `Proizvoda`. Atribut `cascade` je usmeren, tj. primenjuje se samo na jedan kraj veze. Mogla bi se dodati i opcija `cascade="save-update"` u vezu <many-to-one> preslikavanja `Aukcija`, ali pošto se aukcije prave posle proizvoda, to nema smisla.

JPA takođe podržava kaskadiranje stanja instance entiteta u asocijacijama:

```
public class Proizvod {
...
@OneToMany(cascade = {CascadeType.PERSIST, CascadeType=MERGE},
mappedBy="proizvod")
private Set<Aukcija> aukcije = new HashSet<Aukcija>();
...
}
```

Sada se kod koji povezuje i čuva `Proizvod` i `Aukciju` može uprostiti:

```
Proizvod novProizvod = new Proizvod();
Aukcija novaAukcija = new Aukcija();
novProizvod.addAukcija(novaAukcija);
session.save(novProizvod);
```

Čini se razumnim da brisanje proizvoda za sobom povlači i brisanje svih njegovih aukcija; to i označava kompozicija (popunjeni romb) u UML dijagramu. Sa trenutnim podešavanjima, to bi zahtevalo pisanje sledećeg koda:

```
for (Aukcija aukcija :  item.getAukcije()) {
    aukcija.remove(); // uklanjanje reference iz kolekcije
    session.delete(aukcija); //uklanjanje iz baze podataka
}
session.delete(proizvod); // na kraju, brisanje proizvoda
```

Hibernate nudi opciju kaskadiranja i za ovu namenu:

```
<set name="aukcije" inverse="true" cascade="save-update, delete">
   ...
```

Isti kod za brisanje proizvoda i svih njegovih aukcija se svodi na sledeće:

```
session.delete(proizvod);
entityManager.remove(proizvod);
```

Sa JPA anotacijama, preslikavanje bi izgledalo ovako:

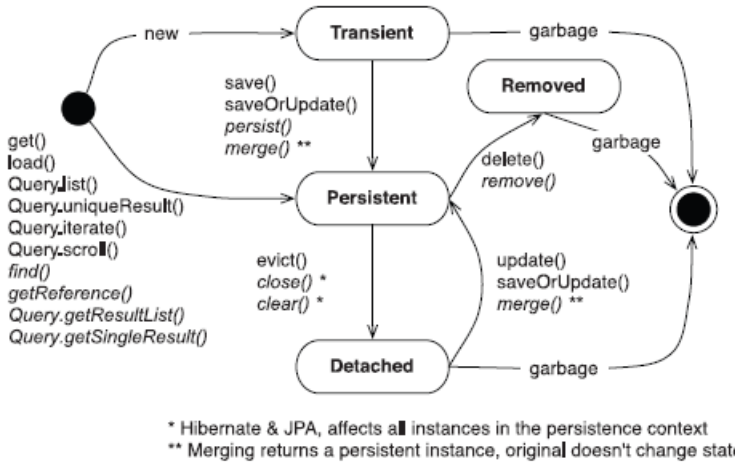
```
public class Proizvod {
...

@OneToMany(cascade = {CascadeType.PERSIST, CascadeType=MERGE,
CascadeType.REMOVE}, mappedBy="proizvod")
private Set<Aukcija> aukcije = new HashSet<Aukcija>();
...
}
```

Ako su reference na objekte `Aukcija` deljene (npr. klasa `User` bi mogla da ima kolekciju referenci na svoje aukcije), nećete moći prosto da obrišete proizvod i sve njegove aukcije bez prethodnog uklanjanja referenci na aukcije iz objekata `User`. U tom slučaju bio bi narušen integritet baze, pa vam ne preostaje ništa drugo nego da "jurite" pokazivače, tj. da uklonite ručno sve reference na aukcije iz svih objekata `User` pre nego što obrišete aukcije i na kraju proizvod.

Rad sa objektima

Životni ciklus perzistiranih objekata

Tokom svog životnog ciklusa, perzistirani objekat prolazi kroz različite faze. Takođe, on ima svoj kontekst perzistencije (engl. *persistence context*), koji možemo zamisliti kao keš koji pamti sve izmene stanja objekata u određenoj jedinici rada. Objekti kojima upravlja Hibernate mogu se naći u jednoj od četiri faze; promenama stanja upravlja objekat `Session`.



Tranzijentni (engl. *transient*) objekti su oni koji nisu odmah perzistentni, već su instancirani operatorom `new`. Oni su u tranzijentnom (prelaznom) stanju jer nisu povezani ni sa jednim zapisom u bazi, pa je njihovo stanje izgubljeno čim ih više ne referencira neki drugi perzistentni objekat. Da bi prešla u perzistentno stanje, tranzijentna instanca mora da pozove sesiju ili da se na nju napravi refrenca iz neke druge instance u perzistentnom stanju.

Perzistentna (engl. *persistent*) je instanca entiteta koja ima identitet u bazi podataka, tj. ima svoj primarni ključ. Perzistentne instance su uvek povezane sa kontekstom perzistencije; Hibernate ih kešira i prepoznaje kada se oni u aplikaciji izmene.

Objekat je uklonjen (engl. *removed*) ako je zakazano njegovo brisanje na kraju jedinice rada, ali njime još upravlja kontekst perzistencije dok se jedinica rada ne završi. Drugim rečima, objekat u uklonjenom stanju ne bi trebalo ponovo koristiti zato što će biti obrisan iz baze čim se jedinica rada završi.

"Otkačeni" (engl. *detached*) su objekti za koje Hibernate više ne garantuje da im je stanje sinhronizovano sa bazom podataka. Oni i dalje sadrže perzistirane podatke i sa njima možete da nastavite da radite i da ih menjate. Međutim, u nekom trenutku ćete morati da perzistirate te izmene, odnosno da vratite otkačen objekat u perzistentno stanje. Za to postoje dva načina, ponovno zakačinjanje (engl. *reattachment*) i spajanje (engl. *merging*) o kojima će biti reči kasnije.

Kontekst perzistencije (engl. *persistence context*) je keš instanci entiteta kojima Hibernate upravlja; to nije nešto što ćete videti u aplikaciji. Obično kažemo da jedna sesija (objekat `Session`) ima jedan unutrašnji kontekst perzistencije. Taj kontekst je koristan između ostalog iz sledećih razloga:

- Hibernate može automatski da prepozna da je objekat izmenjen u jedinici rada (engl. *dirty checking*) i da optimizuje ažuriranje baze (engl. *transactional write-behind*)
- Hibernate može da koristi kontekst perzistencije kao keš prvog nivoa
- Hibernate može da garantuje doseg (engl. *scope*) Javinog identiteta objekata.

Perzistentnim instancama se upravlja u kontekstu perzistencije, tj. njihovo stanje se sinhronizuje na kraju jedinice rada. ORM softver ima strategiju kojojm prepoznaje koji perzistirani objekti su izmenjeni u aplikaciji (engl. *dirty checking*); objekat čije izmene još nisu propagirane u bazu podataka zove se "prljav" (engl. *dirty*). To stanje nije vidljivo u aplikaciji; uz svojstvo transakcionog upisa Hibernate propagira izmene stanja objekta što kasnije moguće, ali skriva detalje o tome od aplikacije. Takođe, ako se traži da se objekat učita na osnovnu primarnog ključa, Hibernate će prvo taj objekat potražiti u kontekstu perzistencije tekuće jedinice rada; ako ga tamo ne nađe, tek onda će ga potražiti u bazi podataka.

Na početku jedinice rada, aplikacija dobija instancu objekta `Session` preko objekta
`SessionFactory`:

```
Session session = sessionFactory.openSession();
Transaction transaction = session.beginTransaction();
```

U ovom trenutku, inicijalizovan je i kontekst perzistencije koji će upravljati svim objektima sa kojima će se raditi u sesiji. Aplikacija može da ima nekoliko objekata `SessionFactory` ako radi sa različitim bazama podataka; ipak, imajte u vidu da je kreiranje objekta `SessionFactory` veoma skupo, za razliku od objekta `Session` čije je kreiranje jeftino (objekat `Session` čak ni ne dobija JDBC objekat `Connection` dok to ne postane neophodno).

Tranzijentni objekat se perzistira pozivom metode `save()`:

```
Proizvod proizvod = new Proizvod();
proizvod.setIme("Playstation 3");
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
Serializable proizvodId = session.save(proizvod);
tx.commit();
session.close();
```

Izmene perzistentnih objekata sinhronizuju se sa bazom kada se pozove `commit()` za transakciju; kažemo da se tada desi flush (metoda `flush()` za transakciju se može pozvati i ručno). Na kraju, zatvara se sesija i završava se kontekst perzistencije. Referenca na objekat proizvod je referenca na objekat u otkačenom stanju. Sve što se dešava između naredbi `session.beginTransaction()` i `tx.commit()` se dešava u jednoj transakciji. Ako transakcija ne uspe, rollback se odvija u bazi podataka, ali ne i u memoriji.

Objekat `Session` se koristi i za slanje upita u bazu podataka i za dohvatanje postojećih perzistiranih objekata. Za tu vrstu upita postoje dva specijalna metoda: `get()` i `load()`:

```
Session session = sessionFactory.openSession();
Transaction tx = session.beginTransaction();
Proizvod proizvod = (Proizvod) session.load(Proizvod.class, new
   Long(1234));
//Proizvod proizvod = (Proizvod) session.get(Proizvod.class, new
   Long(1234));
tx.commit();
session.close();
```

Objekat `proizvod` je u perzistiranom stanju, a čim se kontekst perzistencije zatvori, u otkačenom stanju. Jedna od razlika između metoda `get()` i `load()` je u tome kako javljaju da instanca ne može da se nađe. Ako zapis sa zadatim identifikatorom ne postoji u bazi podataka, `get()` vraća `null`, a `load()` baca izuzetak `ObjectNotFoundException`. Druga, važnija razlika je u tome što `load()` vraća proksi objekta, odnosno njegov šablon, bez upita u bazi podataka. Metoda `load()` uvek prvo pokušava da vrati proksi objekta, a samo ako se objekat već nalazi u kontekstu perzistencije vraća inicijalizovanu instancu objekta. Metoda `get()` s druge strane uvek šalje upit u bazu podataka. Metoda `load()` je korisna u situacijama kada je objekat potreban samo za povezivanje, npr. `komentar.setZaAukciju(proizvod)`. Kada u bazi treba da se perzistira `komentar`, potrebna je samo vrednost spoljnog ključa `proizvoda` koja se čuva u tabeli KOMENTAR; za to je sasvim dovoljan proksi objekta, tj. šablon koji liči na pravi objekat.

Svaki objekat koji vrate metode `get()`, `load()` ili neki entitet za koji se šalje upit već je povezan sa tekućom sesijom i kontekstom perzistencije; on se može menjati, a njegovo stanje se sinhronizuje sa bazom. Ako hoćemo da ga vratimo u tranzijentno stanje, pozvaćemo metod `delete()`:

```
Proizvod proizvod = (Proizvod) session.load(Proizvod.class, new
   Long(1234));
session.delete(proizvod);
```

Nakon poziva metoda `delete()` objekat je u uklonjenom stanju i sa njim više ne treba raditi; SQL naredba DELETE izvršiće se tek kada se kontekst perzistencije sesije sinhronizuje sa bazom na kraju jedinice rada. Menjanje `proizvoda` nakon

što je sesija zatvorena nema uticaja na njegovo stanje u bazi podataka; on je tada otkačen. Ako želite da snimite izmene otkačenog objekta, morate da ga ponovo zakačite. Otkačena instanca se može povezati sa novom sesijom (i njenim kontekstom perzistencije) pozivanjem metode `update()`, koja uvek izvršava SQL naredbu UPDATE:

```
Session drugaSesija = sessionFactory.openSession();
Transaction tx = drugaSesija.beginTransaction();
drugaSesija.update(proizvod);
proizvod.setDatum(...);
tx.commit();
session.close();
```

Ako ste sigurni da niste menjali otkačenu instancu, možete pozvati metodu `lock()` u drugoj sesiji i njenom kontekstu perzistencije; sve izmene pre poziva metode `lock()` neće biti propagirane u bazu podataka i ovaj metod neće generisati SQL naredbu UPDATE. Ako otkačeni metod želite da učinite tranzijentnim, tj. da ga obrišete, dovoljno je da pozovete metodu `delete()`; ona povezuje objekat sa drugom sesijom i zakazuje njegovo brisanje, koje se dešava u naredbi `tx.commit()`.

Kontekst perzistencije (keš perzistiranih objekata) možete da ispraznite pozivom metode `session.clear()`, a pojedinačne objekte da uklanjate iz keša pomoću metode `session.evict(object)`. Metoda `session.setReadOnly(object, true)` isključuje proveru da li je objekat izmenjen (dirty checking). Sinhronizacija konteksta perzistencije sa bazom podataka (tzv. flush) može se ručno pozvati metodom `session.flush()`.

HQL (Hibernate Query Language)

Upiti su najzanimljiviji deo rada u alatu Hibernate. Pravilno pisanje složenog upita ponekad zahteva mnogo vremena, a njegov uticaj na performanse može biti iznenađujuće veliki. S druge strane, pisanje upita postaje mnogo lakše sa iskustvom, pa se nešto što se na početku činilo teško rešava korišćenjem naprednih trikova. Programeri koji su navikli da ručno pišu SQL ponekad su zabrinuti da će im ORM alat oduzeti mogućnost izražavanja i fleksibilnost na koju su navikli u jeziku SQL. Međutim, to nije tako; snažne funkcije za pisanje upita u alatu Hibernate omogućuju izvršavanje svih uobičajenih (ili neobičnih) SQL naredbi, ali na objektno orijentisan način, uz korišćenje klasa i njihovih podataka članova.

Hibernate podržava tri načina za izražavanje upita:

- Hibernate Query Language (HQL):

  ```
  session.createQuery("from Kategorija k where k.ime like 'Laptop%'");
  ```

- API za query by criteria (QBC) i query by example (QBE):

```
session.createCriteria(Kategorija.class).add(Restrictions.like("ime",
   "Laptop%") );
```

- Direktan SQL sa automatskim preslikavanjem skupa rezultata u objekte, ili bez njega:

```
session.createSQLQuery("select {k.*} from KATEGORIJA {c} where IME
   like 'Laptop%'").
addEntity("k", Kategorija.class);
```

Priprema upita

Da bi se napravila nova instanca Hibernate objekta `Query`, može se pozvati metoda `createQuery()` ili `createSQLQuery()` objekta `Session`. Metoda `createQuery()` priprema HQL upit:

```
Query hqlQuery = session.createQuery("from Korisnik");
```

Hibernate vraća nov objekat `Query` koji se može iskoristiti za zadavanje tačnog načina izvršavanja upita. U ovom trenutku u bazu podataka nije poslata nijedna SQL naredba.

Često korišćena tehnika je paginacija rezultata. Korisnicima se rezultati zadate pretrage (na primer, određenih `Proizvoda`) prikazuju na strani koja prima ograničen podskup rezultata (recimo, 10), a korisnici zatim ručno idu napred i nazad. Hibernate interfejs `Query` podržava paginaciju rezultata:

```
Query query = session.createQuery("from Korisnik k order by k.name
   asc");

query.setMaxResults(10);
```

Poziv metode `setMaxResults(10)` ograničava skup rezultata upita na prvih deset objekata (zapisa) koje vraća baza podataka. Primetite da ne postoji standardan način za izražavanje paginacije u jeziku SQL, ali Hibernate zna trikove kojima to postiže za svaku pojedinačnu bazu podataka!

Hibernate takođe podržava povezivanje parametara tokom izvršavanja upita (engl. *runtime binding*). Kada ono ne bi postojalo, morali bismo da pišemo sledeći kod:

```
String queryString = "from Proizvod p where p.opis like '" + search
   + "'";

List result = session.createQuery(queryString).list();
```

Osim što je ružan, ovakav kod ne bi trebalo nikad pisati zato što bi zlonamerni korisnik mogao da u kao vrednost za pretragu opisa proizvoda unese

```
foo' and pozoviNekuProceduru() and 'bar' = 'bar
```

Ako napišete upit sličan ovome, otvarate veliku rupu u bezbednosti aplikacije zato što omogućujete izvršavanje proizvoljnog koda u bazi podataka. Ovaj bezbednosni propust zove se *SQL injection*, no srećom postoji jednostavna tehnika kojom se izbegava. Hibernate podržava tehniku imenovanih parametara (named parameters):

```
String queryString = "from Item item where item.description like
   :search";
```

Dvotačka iza naziva parametra označava imenovani parametar. Zatim treba povezati vrednost sa parametrom `search`:

```
Query q = session.createQuery(queryString).
    setString("search", searchString);
```

Pošto je `searchString` string promenljiva kojoj vrednost dodeljuje korisnik, može se pozvati metoda `setString()` interfejsa `Query` da bi se ona povezala sa imenovanim parametrom (`:search`). Ovaj kod je čistiji, mnogo bezbedniji i efikasniji jer se kompajlirana SQL naredba može ponovo iskoristiti ako se u njoj promene samo povezani parametri.

Interfejs `Query` osim metode `setString()` nudi slične korisne metode za povezivanje argumenata većine tipova koje Hibernate podržava: od `setInteger()` preko `setTimestamp()` do `setLocale()`. Korišćenje ovih metoda nije obavezno, jer se uvek možete osloniti na to da će opšta metoda `setParameter()` automatski prepoznati tip parametra. Jedina pomoć metodi `setParameter()` potrebna je kada je reč o datumima. Koristan metod je i `setEntity()`, koji omogućuje korišćenje nekog perzistentnog entiteta (mada je metoda `setParameter()` i u ovom slučaju dovoljno pametna):

```
session.createQuery("from Proizvod proizvod where proizvod.prodavac
   = :prodavac").
    setEntity("prodavac", instancaProdavca);
```

Često se pojavljuju i upiti sa više imenovanih parametara:

```
String queryString = "from Item item where item.description like
   :search and item.date > :minDate";

Query q = session.createQuery(queryString).
    setString("search", searchString).setDate("minDate", mDate);
```

Metode za povezivanje parametara interfejsa `Query` su null-bezbedne. Zbog toga se može slobodno napisati sledeća naredba:

```
session.createQuery("from Korisnik as k where k.korisnickoIme =
   :ime").setString("ime", null);
```

Međutim, rezultat ovog koda skoro sigurno neće biti ono što smo želeli da postignemo. Dobijeni SQL kod će sadržati poređenje tipa `KORISNICKO_IME = null`, što uvek daje null u SQL logici. Zato umesto toga treba da se koristi null operator:

```
session.createQuery("from Korisnik as k where k.korisnickoIme is
   null");
```

Izvršavanje upita

Pošto smo pripremili objekat `Query`, upit je spreman za izvršavanje da bi se rezultati učitali u memoriju. Postupak smeštanja celog skupa rezultata u memoriju nazivamo listanje. Za izvršavanje upita najčešće se koristi metoda `list()` interfejsa `Query`; ona vraća rezultate kao `java.util.List`:

```
List result = myQuery.list();
```

U nekim slučajevima znamo da će rezultat upita biti samo jedna instanca, npr. ako tražimo aukciju sa najvišom cenom za određeni proizvod. U tom slučaju treba koristiti metodu `uniqueResult()`:

```
Aukcija maxAukcija = (Aukcija) session.createQuery("from Aukcija
   a order by a.iznos desc").setMaxResults(1).uniqueResult();
```

Ako upit vrati više od jednog objekta, biće bačen izuzetak, a ako ne nađe ništa, metoda će vratiti null.

Osnovni HQL upiti

Izbor Najjednostavniji HQL upit je izbor iz jedne perzistentne klase:

```
from Proizvod
```

Ovaj upit generiše sledeći SQL:

```
select p.PROIZVOD_ID, p.IME, p.OPIS, ...  from PROIZVOD p
```

Obično se u naredbi izbora dodeljuje alijas klasi koja se koristi da bi se on referencirao u drugim delovima upita:

```
from Proizvod as proizvod
```

Ključna reč `as` je uvek opciona, pa je prethodni upit isti kao:

```
from Proizvod proizvod
```

Ograničenja Obično ne tražimo sve instance određene klase, već zadajemo određena ograničenja za objekte koje upit treba da vrati; za to se koristi klauzula WHERE. Tipična klauzula WHERE koja ograničava rezultate izgleda ovako:

```
from Korisnik k where k.email = 'user@hibernate.org'
```

Primetite da se ograničenje za email izražava u terminima podatka člana klase `Korisnik` (u ovom slučaju to je `email`) i da se koristi objektno orijentisana notacija. SQL koji generiše ovaj upit je:

```
select k.KORISNIK_ID, k.IME, k.PREZIME, k.KORISNICKO_IME, k.EMAIL
   from KORISNIK k where k.EMAIL = 'user@hibernate.org'
```

U izraze se mogu uključiti konstante navedene pod apostrofima. Često korišćene konstante u HQL-u su TRUE and FALSE:

```
from Proizvod p where p.aktivan = true
```

HQL podržava osnovne operatore za poređenje kao SQL, na primer:

```
from Aukcija a where a.iznos between 1 and 10
```

```
from Aukcija a where a.iznos > 100
```

```
from Korisnik k where k.email in ('foo@bar', 'bar@foo')
```

Pošto SQL baze podataka implementiraju ternarnu logiku[2], provera null vrednosti zahteva posebnu pažnju. HQL podržava operator IS [NOT] NULL:

```
from Korisnik k where k.email is null //daje sve korisnike koji
   nemaju email
```

```
from Proizvod p where p.uspesnaAukcija is not null //vraća sve prodate
   proizvode
```

Operator [NOT] LIKE podržava korišćenje džoker znakova, koji su isti kao u SQL-u: % i _ (procenat zamenjuje proizvoljan niz znakova, a _ jedan znak).

```
from Korisnik k where k.ime like 'G%'
```

Logički operatori (i zagrade za grupisanje) koriste se za kombinovanje izraza:

```
from Korisnik k where (k.ime like 'G%' and k.prezime like 'K%' )
   or k.email in ('foo@hibernate.org', 'bar@hibernate.org')
```

HQL podržava i upite sa kolekcijama, na primer:

```
from Proizvod p where p.aukcije is not empty
```

Ovaj upit vraća sve instance `Proizvoda` koje u svojoj kolekciji aukcije imaju elemente. Ovako se izražava zahteva da određeni element bude u kolekciji:

```
from Proizvod p, Kategorija k where p.id = '123' and p member of
   k.proizvodi
```

Ovaj upit vraća instancu `Proizvoda` sa primarnim ključem '123' i sve instance `Kategorija` sa kojim je taj proizvod povezan.

HQL podržava i pozivanje SQL funkcija; tipičan primer je izraz koji sadrži veličinu kolekcije:

```
from Proizvod p where size(p.aukcije) > 3
```

[2]Zapis se uključuje u SQL skup rezultata ako i samo ako WHERE klauzula ima vrednost true. U Javi, notNullObjekat==null daje false, a null==null vraća true. U SQL-u, NOT_NULL_POLJE= null i null=null vraćaju null, a ne true. Zbog te tzv. ternarne logike SQL podržava poseban operator, IS NULL, koji služi za testiranje da li je određena vrednost null.

Uređivanje rezultata upita HQL podržava klauzulu ORDER BY, slično kao SQL. Sledeći upit vraća sve korisnike poređane po korisničkim imenima:

```
from Korisnik k order by k.korisnickoIme
```

Rastući ili opadajući redosled označava se sa asc ili desc:

```
from Korisnik k order by k.korisnickoIme desc
```

Može se uređivati i po nekoliko svojstava:

```
from Korisnik k order by k.prezime asc, k.ime asc
```

Eliminisanje duplikata Ako znamo da u bazi postoje duplikati zapisa, upotrebićemo ključnu reč distinct:

```
select distinct proizvod.opis from Proizvod proizvod
```

Ovim upitom eliminisaćemo duplikate iz liste opisa `Proizvoda`.

Projekcija Razmotrimo sledeći Hibernate upit:

```
from Proizvod p, Aukcija a
```

Ovaj upit daje uređene parove instanci `Proizvod` i `Aukcija` kao listu `Object[]`. Na poziciji 0 je `Proizvod`, a indeks 1 je `Aukcija`. Pošto se radi o Dekartovom proizvodu tabela, rezultat sadrži sve moguće kombinacije zapisa `Proizvod` i `Aukcija`. Očigledno je da takav upit nije naročito koristan.

```
Query q = session.createQuery("from Proizvod p, Aukcija a");
Iterator parovi = q.list().iterator();
while (parovi.hasNext() ) {
    Object[] par = (Object[]) parovi.next();
    Proizvod proizvod = (Proizvod) par[0];
    Aukcija a = (Aukcija) par[1];
}
```

Ukrštanje tabela Mogućnost ukrštanja podataka iz različitih tabela je jedna od osnovnih prednosti relacionog modela podataka. Ukrštanje tabela (engl. *table join*) takođe omogućuje dohvatanje nekoliko povezanih objekata i kolekcija u jednom upitu. Ukrštanje se koristi za kombinovanje podataka u dve (ili više) relacija. Na primer, mogu se kombinovati podaci iz tabela `PROIZVOD` i `AUKCIJA` kao što je prikazano na slici.

PROIZVOD

PROIZVOD_ID	OPIS	...
1	Proizvod jedan	...
2	Proizvod dva	...
3	Proizvod tri	...

AUKCIJA

AUKCIJA_ID	PROIZVOD_ID	IZNOS
1	1	99.00
2	1	100.00
3	1	101.00
4	2	4.99

Ono što se najčešće podrazumeva kada se pomene ukrštanje u kontekstu SQL baza podataka je tzv. *(levi) inner join* koji je najvažniji od svih tipova kombinovanja i najlakši za razumevanje. Ako ukrstimo tabele `PROIZVOD` i `AUKCIJA` koristeći *inner join*, uz korišćenje njihovih zajedničkih atributa (polja `PROIZVOD_ID`), dobićemo proizvode i aukcije u novoj tabeli sa rezultatima koja će sadržati samo proizvode za koje postoje aukcije.

```
select p.*, a.* from PROIZVOD p inner join AUKCIJA a on
p.PROIZVOD_ID = a.PROIZVOD_ID
```

PROIZVOD_ID	OPIS	...	AUKCIJA_ID	PROIZVOD_ID	IZNOS
1	Proizvod jedan	...	1	1	99.00
1	Proizvod jedan	...	2	1	100.00
1	Proizvod jedan	...	3	1	101.00
2	Proizvod dva	...	4	2	4.99

Ako želimo sve proizvode uz NULL vrednosti u slučajevima kada nema odgovarajućih aukcija, upotrebićemo *(levi) outer join*. U tom slučaju, svaki red u (levoj) tabeli `PROIZVOD` koji ne zadovoljava uslov spajanja takođe će biti uključen u rezultat, sa NULL vrednostima svih kolona tabele `AUKCIJA`.

select p.*, a.* from PROIZVOD p left outer join AUKCIJA a on p.PROIZVOD_ID = a.PROIZVOD_ID

PROIZVOD_ID	OPIS	...	AUKCIJA_ID	PROIZVOD_ID	IZNOS
1	Proizvod jedan	...	1	1	99.00
1	Proizvod jedan	...	2	1	100.00
1	Proizvod jedan	...	3	1	101.00
2	Proizvod dva	...	4	2	4.99
3	Proizvod tri	...	NULL	NULL	NULL

Desni outer join pronalazi sve aukcije i null ako aukcija nema proizvoda, a takav upit u ovom slučaju nema smisla. *Desni outer join* se retko koristi jer programeri uvek razmišljaju sleva udesno i tabelu koja upravlja upitom stavljaju na početak.

U nastavku ćemo se pozabaviti različitim načinima kako se može izraziti ukrštanje u alatu Hibernate.

Implicitni join Razmotrimo sledeći HQL:

```
from Aukcija aukcija where aukcija.proizvod.opis like '%Foo%'
```

Ovakim upitom se kreira implicitno ukrštanje u vezama tipa many-to-one kao što je veza između `Aukcije` i `Proizvoda`. Hibernate zna da smo tu vezu preslikali pomoću spoljnjeg ključa `PROIZVOD_ID` u tabeli aukcija i generiše odgovarajući SQL join. Implicitni join nikada ne koristi smer od kolekcije (tj. ne može da se napiše `proizvod.aukcije.iznos`).

Višestruki join je moguć u jednom izrazu. Zamislimo za trenutak da je veza između `Proizvoda` i `Kategorije` tipa many-to-one (umesto many-to-many kakva je zaista), odnosno da proizvod može da pripada samo jednoj kategoriji. Tada bismo mogli da napišemo sledeći upit:

```
from Aukcija aukcija where aukcija.proizvod.kategorija.ime like
   'Laptop%'
```

Ili, na primer, sledeći upit:

```
from Aukcija aukcija where aukcija.proizvod.kategorija.ime like
   'Laptop%' and aukcija.proizvod.uspesnaAukcija.iznos > 100
```

Pošto je broj joina veoma važan za performanse upita, zapitajmo se kako bismo ovaj upit izrazili u jeziku SQL i koliko bi joina on imao? Odgovor na ovo pitanje zahtevao bi izvesno vreme:

```
select ...  from AUKCIJA A inner join PROIZVOD P on A.PROIZVOD_ID
   = P.PROIZVOD_ID inner join KATEGORIJA K on P.KATEGORIJA_ID =
   K.KATEGORIJA_ID inner join AUKCIJA AU on P.USPESNA_AUKCIJA_ID
   = AU.AUKCIJA_ID where K.IME like 'Laptop%' and AU.IZNOS > 100
```

Join izražen u klauzuli FROM Pretpostavimo da pravimo upit nad objektima klase `Proizvod`. Dva su razloga zbog kojih bismo želeli da ga kombinujemo sa `Aukcijama`. Možda želimo da ograničimo proizvode koje upit vraća na osnovu nekog kriterijuma koji bi se primenio na njihove aukcije. Na primer, možda nam trebaju svi `Proizvodi` čije su cene na aukciji prešle 100 USD; to bi zahtevalo inner join. U tom slučaju nismo zainteresovani za proizvode koji nemaju aukcije. S druge strane, možemo u principu biti zainteresovani za `Proizvode`, ali nam treba outer join zato što želimo da dobijemo sve `Aukcije` za `Proizvode` u istoj SQL naredbi.

U prvom slučaju, upotrebićemo inner join za dobijanje instanci `Proizvoda` i ograničavanje na proizvode čija je cena na aukciji premašila određen iznos:

```
from Proizvod p join p.aukcije a where p.opis like '%Foo%' and a.iznos
   > 100
```

Ovaj upit dodeljuje alijas `p` entitetu `Proizvod` i alijas `b` pridruženim aukcijama. Zatim se oba alijasa koriste za izražavanje ograničenja u klauzuli WHERE.

```
from Proizvod p join p.aukcije a
```

Ovaj upit vraća sve kombinacije pridruženih `Aukcija` i `Proizvoda` kao uređenih parova. Ako ne želimo `Aukcije` u rezultatima upita, zadaćemo klauzulu SELECT uz alijas koja će projektovati samo objekte koje želimo:

```
select p from Proizvod p join p.aukcije a where p.opis like '%Foo%'
   and a.iznos > 100
```

HQL nudi i alternativnu sintaksu za join kolekcija u FROM klauzuli i dodeljivanje alijasa; to je operator IN():

```
select p from Proizvod p in(p.aukcije) a where p.opis like '%Foo%'
   and a.iznos > 100
```