PT&SCHE

Introduction of part-time and short cycle studies in Serbia

Funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

UNIVERZITET
METROPOLITAN
BEOGRAD

KI203 - JAVA 6: NAPREDNO PROGRAMIRANJE U JAVI

# Skladištenje objekata sa Hibernate ORM

Lekcija 05

PRIRUČNIK ZA STUDENTE

# KI203 - JAVA 6: NAPREDNO PROGRAMIRANJE U JAVI

Lekcija 05

## *SKLADIŠTENJE OBJEKATA SA HIBERNATE ORM*

# UVODNE NAPOMENE

*Najava sadržaja lekcije - Hibernate ORM i nekih izvora informacija o ovom radnom okviru.*

Objektno-orijentisane aplikacije najčešće koriste objekte koji treba da se trajno sačuvaju, tj. da se smeste (uskladište) u trajnu memoriju, kao što je najčešće disk (magnetni, optički). Java nudi JDBC interfejs, tj. metode koje pomažu programerima da programiraju takve operacije sa relacionim bazama podataka, i to nezavisno od proizvođača sistema za upravljanje bazom podataka (RDBMS - Relational Database Management System).

Međutim, prebacivanje objekata u tabele koje koriste relacione baze zahteva poznavanje njene strukture, a kada se ona menja, mora da se menja i deo koda aplikacije. To očigledno pokazuje da postoji prirodna neusklađenost objektnog modela koji koristi aplikacija, i relacionog modela, koji koristi baza podataka. Da bi se ovaj problem jednostavnije rešio, i da bi programeri bili zaštićeni od potrebe da menjaju programski kod kod svake izmene strukture baze podataka, razvijeni su tzv. Object Relational Management (ORM) sistemi koji olakšavaju posao programerima. U ovoj lekciji ćemo prikazati jedan od najpopularnijih ORM sistema - Hibernate.

Prikaz Hibernate-a u ovoj lekciji se najvećim delom zasniva na prikazu datom na sajtu **tutorialspoint.com**: **Hibernate Tutorial** http://www.tutorialspoint.com/hibernate/index.htm

Pored niza knjiga o Hibernate-u (na primer, u izdanju O'Reilly *Harnessing Hibernate* 1st Edition, autora James Elliott, Timothy M. O'Brien, Ryan Fowler) , možete koristiti i sledeće sajtove da saznate više o Hibernate-u

- Hibernate − zvaničan sajt za Hibernate, JBoss Community.
- Hibernate Documentation − Daju se linkovi ka referentnoj dokumentaciji, javadocs i druge literature korisnika Hibernate-a
- iBATIS − Zvaničan sajt iBATIS, Apache Software Foundation.
- iBATIS for Java − sajt za preuzimanje iBATIS softvera i dokumentacije
- za JDBC: http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136101.html
- za MySQL: http://dev.mysql.com/downloads/connector/j/5.1.html

# OMR - PRESLIKAVANJE OBJEKATA U RELACIJE

*OMR vrši konverziju podataka između OO aplikacije i relacione baze podataka na naičin koji nosi više prednosti u odnosu na primenu JDBC interfejsa.*

ORM je skraćenica od Object-Relational Mapping i predstavlja tehniku programiranja kojom se vrši konverzija podataka između relacionih baza podataka i objektno-orijentisanih jezika, kao što su Java, C# i dr. ORM je radni okvir (framework) za mapiranje objektno-orijentisanog Domain modela na tradicionalnim relacionim bazama.

ORM obezbeđuje sledeće prednosti, u odnosu na JDBS:

1. Omogućava programu da pristupa objektima, a ne tabelama u bazi.
2. Sakriva detalje o SQL upitima od OO programa i njegove logike.
3. Baziran je na JDBC, koji je u "pozadini".
4. Nezavisan je od implementacije baze podataka, tj. od proizvođača DBMS
5. Primenjeni entiteti su prilagođeni poslovnom konceptu, a ne strukturi baze podataka
6. Upravljanje transakcijama i automatsko generisanje ključeva.
7. Brz razvoj aplikacija..

Aplikacija koja koristi ORM sadrži četiri entiteta:

- API koji izvršava osnovne CRUD operacije sa objektima trajnih (persistent) klasa
- Jezik ili API za specifikaciju upita bazi koji se odnose na klase i svojstva klasa.
- Konfuigurabilno sredstvo za specifikaciju metapodataka mapiranja.
- Tehniku za interakciju objektima u transakciji radi izvršenja "prljave" (*dirty*) provere, usporenog preuzimanja asocijacija (*lazy association fetching*), i druge funkcije optimizacija.

Postoji nekoliko OMR trajnih radnih okvira (*persistent frameworks*) u Javi. Trajni radni okvir je ORM service koji memoriše i pretražuje objekte u nekoj relacionoj bazi:

- *Enterprise JavaBeans Entity Beans*
- *Java Data Objects*
- *Castor*
- *TopLink*
- *Spring DAO*
- *Hibernate*
- i drugi

# Poglavlje 1

# Problemi mapiranja objekata u relacione tabele

## JDBC I ORM

*JDBC je Java API koji omogućava Java programu da izvršava SQL iskaze i da ima interakciju sa svakom bazom podataka koja koristi SQL*

**Šta je JDBC?**

JDBC je skraćenica od Java Database Connectivity (Veza Jave i baze podataka). JDBC obezbeđuje više aplikacionih interfejsa (API) za razvoj aplikacija u Javi - Java API. JDBC je Java API koji omogućava pristup relacionim bazama iz Java programa. JDBC je Java API koji omogućava Java programu da izvršava SQL iskaze i da ima interakciju sa svakom bazom podataka koja koristi SQL.

JDBC omogućava fleksibilnu arhitekturu za razvoj Java aplikacija koje su nezavisne od proizvođača sistema relacione baze podataka (RDBMS).

| Dobre strane JDBC | Loše strane JDBC |
|---|---|
| •Jasna i jednostavna SQL obrada<br>•Dobre performanse sa mnogo podataka<br>•Vrlo dobar za male aplikacije<br>•Jednostavna sintaksa i lako se uči | •Složena promena kod velikih projekata<br>•Znatan rad programera<br>•Nema učaurenja<br>•Teško je primeniti MVC koncept<br>•Upiti su zavisni od DBMS |

Slika 1.1.1 T1 Dobre i loše strane primene JDBC interfejsa

**Zašto Object Relational Mapping (ORM)?**

Kada radimo sa objektno-orijentisanim sistemima, postoje neusaglašenost između objektnog modela i modela relacione baze podataka. RDBMS predstavljaju podatke u obliku tabele.

Uzmimo na primer, sledeću Java klasu sa odgovarajućim konstruktorom i pridodatom javnom funkcijom:

```
public class Employee {
   private int id;
   private String first_name;
   private String last_name;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
```

```
      this.first_name = fname;
      this.last_name = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public String getFirstName() {
      return first_name;
   }
   public String getLastName() {
      return last_name;
   }
   public int getSalary() {
      return salary;
   }
}
```

# PROBLEM NEUSKLAĐENOSTI OBJEKTNOG I RELACIONOG MODELA

*Pri preuzimanju objekata iz baze i kod uskladištenja objekata u bazu, javljaju se sledeći problemi uvezani za: granularnost, nasleđivanje, identitet, asocijacije i navigaciju.*

Predpostavimo da navedene objekte treba da smestimo u tabelu RDBMS i da ih kasnije vraćamo nazad, iz tabele u OO aplikaciju.

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Prvi problem se javalja kada želimo da promenimo model naše baze podataka, a već smo razvili nekoliko stranica naše aplikacije, jer moramo da menjamo deo onoga što smo ranije uradili. Drugi problem se javlja pri preuzimanju objekata iz baze i kod uskladištenja objekata u bazu. Sledeća tabela pokazuje do kakvih problema može doći.

| Neslaganje | Opis |
|---|---|
| Granularity (granularnost) | Ponekad ćete imati objektni model koji ima više klasa nego što baza ima broj odgovarajućih tabela. |
| Inheritance (nasleđivanje) | RDBMS ne definišu nasleđivanje što je prirodna paradigme u OO programskim jezicima. |
| Identity (identifikacija) | RDBMS za idenifikaciju koristi primarni ključ. Java, pak, za identifikaciju koristi i identičnost objekata (a==b) i jednakost objekata (a.equals.b). |
| Associations (asocijacije) | Objektno-orijentisani jezici predstavljaju asocijacije preko referenci, dok EDBMS predstavlja asocijaciju kao kolonu sa stranim ključem. |
| Navigation (navigacija) | Način pristupanju objketima u Javi i u RDMS su fundamentalno različiti. |

Slika 1.1.2 T2 Neslaganje objektnog i relacionog modela

# ŠTA JE ORM?

## *ORM je tehnika programiranja koja vrši konverziju podataka između relacionih baza podataka i objektno-orijentisanih jezika*

Object-Relational Mapping (ORM) je rešenje za navedene probleme neusklađenosti objektnog i relacionog modela.

ORM je tehnika programiranja koja vrši konverziju podataka između relacionih baza podataka i objektno-orijentisanih jezika, kao što su Java, C# i dr. ORM sistem ima sledeće prednosti u odnosu na JDBC:

1. Dozvoljava kodu aplikacije da pristupi objektima, a ne tabelama baze.
2. Sakriva detalje SQL upita od OO logike.
3. Koristi u pozadini JDBC.
4. Ne zavisi od primenjenog sistema baza podataka (RDBMS)
5. Entiteti se zasnivaju na poslovnim konceptima, a ne na strukturi baze podataka.
6. Upravljanje transakcijama i automatska generacija ključa.
7. Brz razvoj aplokacije.

ORM rešenje sadrži sledeća četiri entiteta, tj. komponente:

1. Jedan API koji odrađuje osnovne CRUD operacije na objektima trajnih klasa.
2. Jezik ili API za specifikaciju upita koji povezuju klase i svojstva klas.
3. Fajl za konfigurisanje, tj. za specifikaciju mapiranja meta podataka.
4. Tehniku za interakciju sa transakcionim objektima radi izvršenja prljave provere (checking), sporog usaglašavnja veza i drugih optimizacionih funkcija.

**Java ORM radni okviri (frameworks):**

Postoji nekoliko radnih okvira u Javi za rad sa uskadištenjem objekata i ORM opcija. Jedan od radnih okvira je ORM servis koji skladišti objekte u relacionu bazu i vraća ih nazad u aplikaciju.

***Java Enterprise Beans***, ***Java Data Objects, Castor***,***Hibernate***, ***TopLink, Spring DAO*** i dr.

## 1.1 Zadaci za samostalni rad

### ZADACI ZA SAMOSTALNI RAD STUDENTA

*Cilj je razumevanje naučenog*

**Zadatak 1:**

Šta predstavlja ORM? Navedite prednosti i nedostatke korišćenja ORM-a.

# Poglavlje 2

# Hibernate ORM - Mapiranje objekata u bazu

## ŠTA JE HIBERNATE?

*Hibernate je softverska komponenta koja se nalazi između Java objekata (u aplikaciji) i servera sa bazom podataka, i koja obavlja celokupan posao vezan za mapirane objekata u relacione tab*

Hibernate je OMR rešenje za Javu koje je u vidu jednog otvorenog radnog okvira za uskladištenje objekata (open source persistent framework) kreirao Gavin King 2001. godine. Hibernate je moćan servis sa visokim performansama, za uskladištenje (trajno memorisanje) objekata i njihovo pretraživanje i dobijanje iz trajne memorije koje može da koristi bilo koja aplikacija u Javi.

Hibeate mapira Java klase u tabele baze podataka i tipove podataka Jave u SQL tipove podataka i oslobađa programera od potrebe da realizuje 95% uobičajenih zadataka pri radu sa smeštanjem i korištenjem objekata u trajnoj memoriji (relacionoj bazi podataka).

Hibernate je softverska komponenta koja se nalazi između Java objekata (u aplikaciji) i servera sa bazom podataka, i koja obavlja celokupan posao vezan za mapirane objelata u relacione tabele u bazi, koristeći odgovarajuće O/R mehanizme i šablone (slika 1)..


Slika 2.1.1 Hibernate kao komponenta između Java objekti i RDBMS

Prednosti primene Hibernate OMR rešenja:

- Hibernate se brine o mapiranju Java klasa u tabele baze podataka upotrebom XML fajlova bez potrebe da se napiše ni jedna linija programskog koda.
- Obezbeđuje jednostavne API za smeštaj i preuzimanje Java objekata iz baze podataka i ka bazi podataka.
- Ako dođe do promene u bazi (Database) ili u nekoj tabeli potrebo je samo da se promeni odgovarajuće svojstvo u XML fajlu.
- Nema potrebe da se radi sa SQL tipovim, te obezbeđuje rad samo sa Java objektima.
- Hibernate ne zahteva korištenje aplikacionog servera.

- Radi sa složenim asocijacijama objekata u bazi podataka.
- Obezbeđuje jedostavne upite podataka.

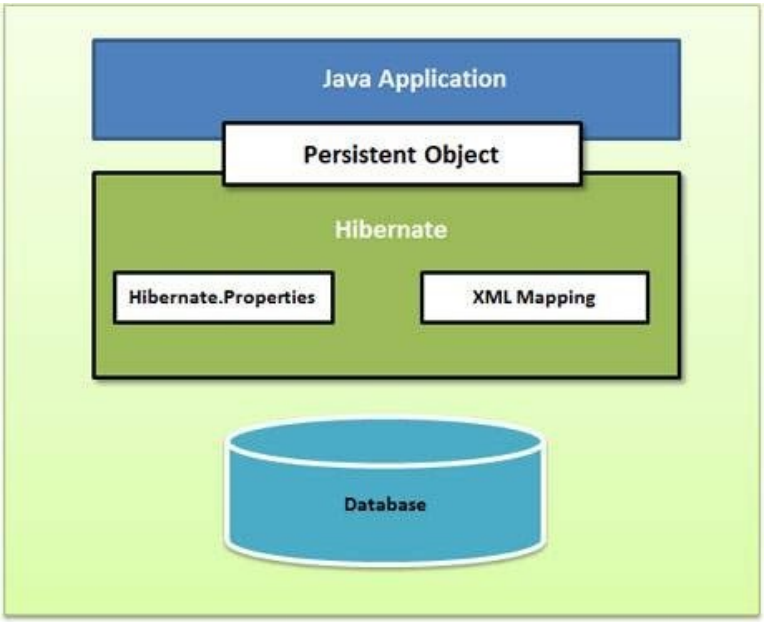
Hibernate radi sa sledećim sistemima relacionih baza podataka (RDBMS): HSQL Database Engine, DB2/NT, MySQL, PostgreSQL, FrontBase, Oracle, Microsoft SQL Server Database, Sybase SQL Server i Informix Dynamic Server,

Hibernate podržva sledeće tehnologije, XDoclet Spring, J2EEEclipse plug-ins i Maven.

# HIBERNATE - ARHITEKTURA

*Hibernate ORM koristi slojevitu arhitekturu te programer aplikacije ne mora da zna i koristi API nižih slojeva.*

Arhitektura Hibernate ORM je slojevita da bi vas izolovala od potrebe da znate API nižih slojeva. Hibernate omogućava upotrebu baze podataka i konfiguracionih podataka da bi obezbedio servise uskladištenja objekata neke aplikacije. Donja slika pokazuje arhitekturu Hibernate aplikacije, sa visokim nivoom apstrakcije:




Slika 2.1.2 Arhitektura Hibernate aplikacije

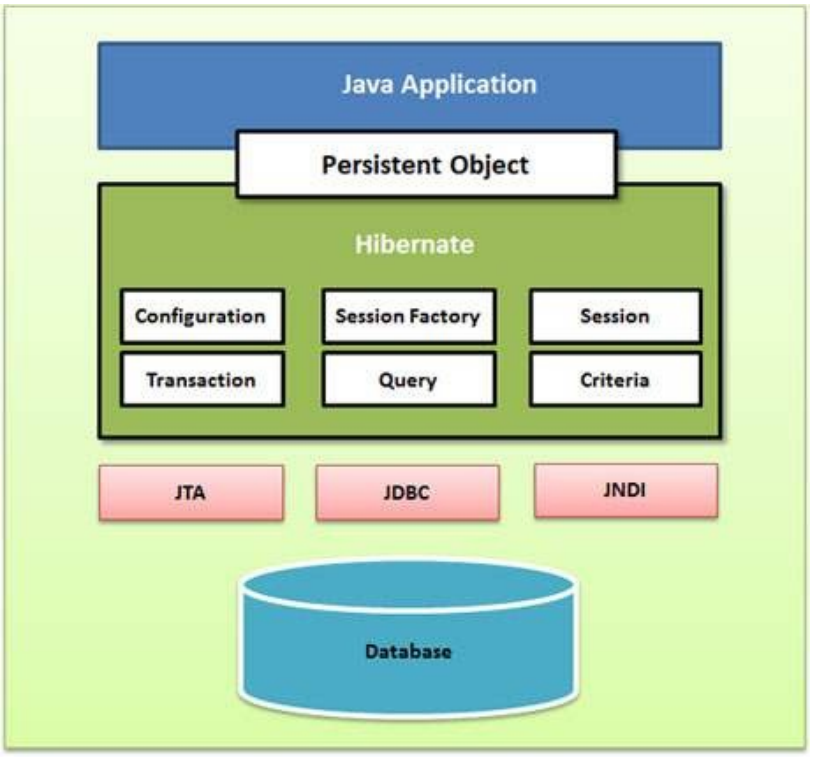
Slika 2 daje malo detaljniji prikaz arhitekture sa nekoliko ključnih Java klasa.

Hibernate upotrebljava različita JAVA API interfejse, kao što su: JDBC, Java Transaction API(JTA), i Java Naming and Directory Interface (JNDI). JDBC obezbeđuje osnovni nivo funkcionalnosti za rad sa relacionom bazom podataka, omogućavajući da Hibernate može da koristi skoro svaku bazu koja koristi JDBC drajver. JNDI i JTA omogućava integraciju Hibernate ORM sa J2EE aplikacionim serverom.

# OBJEKAT CONFIGURATION, CONNECTION I MAPPING

*Configuration objekat kreira SessionFactory objekat koji pak konfiguriše Hibernate za aplikaciju pri čemu koristi konfiguracioni fajl i dozvoljava kreiranje Session objekata.*

Sada ćemo ukratko opisati osnovne klase Hibernate ORM koje su prikazane na slici 3.




Slika 2.1.3 Klase Hibernate ORM

**Objekat Configuration:**

Objekat Configuration je prvi Hibernate objekat koji treba da kreirate u vašoj Hibernate aplikaciji i najčešće se kreira samo jedanput za vreme inicijalizacije aplikacije. Ovaj objekat predstavlja konfiguraciju ili fajl sa svojstvima koju zahteva Hibernate. Configuration objekat ima dve ključne komponente:

**Database Connection**: Ovu komponentu obezbeđuje jedan ili više konfiguracionih fajlova koji podržavaju Hibernate. Ovi fajlovi su ***hibernate.properties*** i ***hibernate.cfg.xml.***
Klasa Mapping Setup: Ova komponenta kreira vezu između Java klasa i tabele baze podataka.

.

# OBJEKTI SESSIONFACTORY, SESSION, TRANSACTION, QUERY I CRITERIA

*Objekat Transaction predstavlja jedinicu rada sa bazom podataka i sa RDMS i podržava transakcije. Query - realizuje upite baze, a Criteria - definiše kriterijume*

**SessionFactory objekat:**

Configuration objekat kreira SessionFactory objekat koji pak konfiguriše Hibernate za aplikaciju pri čemu koristi konfiguracioni fajl i dozvoljava kreiranje Session objekata. SessionFactory radi sa nitima (thread safe) i njih koriste sve niti u jednoj aplikaciji.

SessionFactory je "težak" objekat te se najčešće kreira za vreme startovanja aplikacije i zadržava se za kasniju upotrebu. Potreban vam je po jedan SessionFactory objekat po bazi podataka pri čemu se koristi poseban konfiguracioni fajl. Ako koristite više baza podataka, onda treba da koristiti više SessionFactory objekata.

**Session objekat:**

Session objekat se koristi za dobijanje fizičke veze sa bazom podataka. Session objekat je "lak" objekat koji se kreira uvek kada je potrebna neka interakcija sa bazom podataka. Trajno memorisani objekti se memorišu i prikupljaju primenom Session objekta.

Session objekti ne treba da budu otvoreni dugo vremena jer obično nisu bezbedni za niti, te treba da ih kreirate i uništite samo kada su vam potrebni.

**Transaction objekat:**

Objekat Transaction predstavlja jedinicu rada sa bazom podataka i najveći broj sistema baza podataka (RDBMS) podržava korišćenje transakcija. Radom Hibernate ORM rukovodi menadžer transakcija i Transaction (iz JDBC ili JTA)

To je opcioni objekat i Hibernate aplikacije mogu da ne koriste ovaj interfejs, već da upravljaju transakcijama sopstvenim kodom.

**Query objekat:**

Query objekti upotrebljavaju SQL ili Hibernate Quary Language (HQL) string da bi prikupili podatke iz baze podataka i kreirali objekte. Query objekat se koristi za povezivanje parametara upita, za ograničenje broja rezultata koje vraće upit, i na kraju izvršava upit. .

**Criteria objekat:**

Criteria objekat se koristi za kreiranje i izvršenje objektno-orijentisanih kriterija upita za prikupljanje objekata.

# HIBERNATE - OKRUŽENJE

*Hibernate se preuzima sa http://www.hibernate.org/downloads. Kopirajte hibernate3.jar fajl u vaš CLASSPATH. Morate i preuzeti i više bibliotečkih paketa sa klasama.*

Ovde ćemo objasniti kako da instlirate Hibernate i ostale neophodne pakete, kako bi kreirali razvojno okruženje za Hibernate aplikacije. Koristićemo MySQL bazu u rešavanju primera, a pretpostavka je da imate već instaliranu MySQL bazu. Takođe se predpostavlja da imate instaliranu zadnju verziju Jave.

Pre preuzimanja Hibernate ORM, morate se odlučiti da li želite da ga koristite u Windows ili Unix operativnom sistemu. Poslednju verziju Hibernate ORM možete preuzeti sa: http://www.hibernate.org/downloads ili ***hibernate-distribution-3.6.4.Final***

Posle preuzimanja i dekodiranja (unzip) Hibernate instalacionog fajla, ***hibernate-distribution-3.6.4.Final*** potrebno je da sledite nekoliko sledećih koraka.

1. Podesite svoju CLASSPATH promenljivu kako ne bi imali problem pri kompilaciji.
2. Kopirajte bibliotečke fajlove sa***/lib*** u CLASSPATH i promenite vašu CLASSPATH promenljivu da bi uključiti sve JAR fajlove.
3. Kopirajte ***hibernate3.jar*** fajl u vaš CLASSPATH. Ovaj fajl se nalazi na osnovnom direktorijumu instalacije, kao i primarne JAR koje Hibernate zahteva da bi ispravno radio.

**Hibernate preduslovi:**

Da bi Hibernate mogao da radi, morate da preuzmete više bibliotečkih fajlova sa paketima klasa. Potrebni su vam sledeći paketi u vašem CLASSPATH:

1. dom4j - XML parsing www.dom4j.org/
2. Xalan - XSLT Processor http://xml.apache.org/xalan-j/
3. Xerces - The Xerces Java Parser http://xml.apache.org/xerce
4. cglib - Appropriate changes to Java classes at runtime http://cglib.sourceforgelog4j -
5. Logging Faremwork http://logging.apache.org/log4j
6. Commons - Logging, Email etc. http://jakarta.apache.org/commons
7. SLF4J - Logging Facade for Java http://www.slf4j.org

# HIBERNATE - KONFIGURISANJE

*Hibernate se konfiguriše određivanjem vrednosti njegovih svojstava. Svojstva se definišu za slučaj korišćenja baze podataka ili JNDI aplikacionog servera*

Hibernate zahteva da mu saopštite gde može da nađe informaciju o mapiranju koja definiše ko je vaša Java klasa povezana sa tabelama u bazi podataka. Takođe, on zahteva da mu se podese određeni parametri konfiguracije u vezi sa bazom podataka i drugim odgovarajućim parametrima. Sve ove informacije se obično nalaze u obliku standardnog fajla sa Java svojstvima koji se naziva ***hibernate.properties*** ili u obliku XML fajla pod nazivom ***hibernate.cfg.xml.***

U sledećem primeru ćemo koristiti hibernate.cfg.xml za specificiranje svojstava Hibernate ORM. Ako svojstva imaju standardne vrednosti, ne morate ih posebno navoditi. Fajl sa stavlja u osnovni direktorijum (root directory) vašeg CLASSPATH definisanog za vašu aplikaciju.

**Svojstava Hibernate-a:**

Nadalje navodimo listu važnih svojstava sa kojima možete da konfigurišete svojstva Hiberbate-a u odnosu na vašu bazu podataka.

1. hibernate.dialect - Ovo omogućava da Hibernate generiše odgovarajuće SQL isntrukcije za određenu bazu podataka. .
2. hibernate.connection.driver_class - to je JDBC klasa
3. hibernate.connection.url - URL u JDBC klasi za određenu bazu.
4. hibernate.connection.username - naziv baze podataka
5. hibernate.connection.password - lozinka za pristup bazi podataka
6. hibernate.connection.pool_size - ograničava broj veza Hibernate-a sa pulom veza baze podataka.
7. hibernate.connection.autocommit - omogućava automatsko JDBC povezivanje

Ako upotrebljavate bazu podataka sa aplikacionim serverom i JNDI , onda morate da konfiguraciju izvvršite za sledeća svojstva

1. hibernate.connection.datasource - JNDI ime aplikacionog servera koji koristi vaša aplikacija .
2. hibernate.jndi.class - početni InitialContext klasa za JNDI
3. hibernate.jndi.<JNDIpropertyname> - prebacuje JNDI svojstva u JNDI InitialContext
4. hibernate.jndi.url - daje URL za JNDI
5. hibernate.connection.username - korisničko ime baze podataka
6. hibernate.connection.password - lozinka za pristup bazi

# HIBERNATE SA MYSQL BAZOM PODATAKA

*U slučaju korišćenja MySQL baze, potrebno je defisati svojstva u konfiguracionom fajlu hibernate.cfg.xml koji se postavlja u osnovni direktorijum definisanog za vašu aplikaciju.*

Potrebno je da se kreira konfiguracioni fajl ***hibernate.cfg.xml*** i njegovo postavljanje u osnovni direktorijum defisan za vašu aplikaciju (classpath). Potrebno je da proverite da li ste obezbedili **testdb** bazu vaše MySQL baze podataka i da imate korisnički test za pristup bazi.

XML konfiguracioni fajl mora da bude u saglasnosti sa Hibernate 3 Configuration DTD, koji se može preuzeti sa sajta: http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>

   <!-- Assume test is the database name -->
   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/test
   </property>
   <property name="hibernate.connection.username">
      root
   </property>
   <property name="hibernate.connection.password">
      root123
   </property>

   <!-- List of XML mapping files -->
   <mapping resource="Employee.hbm.xml"/>

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

Ovaj konfiguracioni fajl koristi ***<mapping>*** tagove koji odgovaraju fajlu za ***hibernate-mapping*** fajl. Dole se daje lista svojstava za važnije RDBMS: DB2 org.hibernate.dialect.DB2Dialect
**HSQLDB** org.hibernate.dialect.HSQLDialect **HypersonicSQL** org.hibernate.dialect.HSQLDialect
**Informix** org.hibernate.dialect.InformixDialect **Ingres** org.hibernate.dialect.IngresDialect
**Interbase** org.hibernate.dialect.InterbaseDialect
**Microsoft SQL Server 2000** org.hibernate.dialect.SQLServerDialect
**Microsoft SQL Server 2005** org.hibernate.dialect.SQLServer2005Dialect
**Microsoft SQL Server 2008** org.hibernate.dialect.SQLServer2008Dialect **MySQL** org.hibernate.dialect.MySQLDialect
**Oracle** (sve verzije) org.hibernate.dialect.OracleDialect
**Oracle 11g** org.hibernate.dialect.Oracle10gDialect
**Oracle 10g** org.hibernate.dialect.Oracle10gDialect
**Oracle 9i** org.hibernate.dialect.Oracle9iDialect
**PostgreSQL** org.hibernate.dialect.PostgreSQLDialect **Progress** org.hibernate.dialect.ProgressDialect
**SAP DB** org.hibernate.dialect.SAPDBDialect
**Sybase** org.hibernate.dialect.SybaseDialect
**Sybase Anywhere** org.hibernate.dialect.SybaseAnywhereDialect

# HIBERNATE - SESSION OBJEKAT

*Glavna funkcija Session objekta je da kreira, čita i briše operacije nad objektima klasa koje se mapiraju.*

Objekat Session se koristi za uspostavljanje fizičke veze sa bazom podataka. Session objekat je "lak" jer može da se kreira uvek kada je potrebna interakcija sa bazom podataka. Trajni objekti se skladište i prikupljaju se preko Session objekta.

Session objekti ne bi trebalo da budu otvoreni dugo, samo onoliko koliko je nužno, jer obično nisu nitno bezbedni (thread safe), te se kreiraju samo kada su potrebni, i uništavaju odmah po prestanku potrebe. Glavna funkcija Session objekta je da kreira, čita i briše operacije nad objektima klasa koje se mapiraju. Session objekti mogu da imaju jedno od sledeća tri stanja:

1. ***transient:*** Novi objekat trajne klase koja nije povezana sa Session objektom i nema predstavnika u bazi podataka, i nema ni identifikacionu vrednost smatra se privremenim od strane Hibernate ORM.
2. ***persistent:*** Privremeni objekat postaje trajan njegovim povezivanjem sa ***Session*** objektom. Trajan objekt ima svog predstavnika u bazi podataka, ima identifikatorsku vrednost i vezu sa Session objektom.
3. ***detached:*** Po zatvaranju ***Session*** objekta, objekt postaje odvojen

Session objekat je ***serializable*** ako je i njegova trajna klasa ***serializable***. Tipična transakcija bi izgledala ovako:.

```
Session session = factory.openSession();
Transaction tx = null;
try {
   tx = session.beginTransaction();
   // do some work
   ...
   tx.commit();
}
catch (Exception e) {
   if (tx!=null) tx.rollback();
   e.printStackTrace();
}finally {
   session.close();
}
```

# METODI INTERFEJSA SESSION

*Session interfejs obezbeđuje puno metoda koji podržavaju sesije, SQL upite, filtriranje upita, transakcije, definisanje kriterijuma za pretraživanje objekata, dobijanje trajnog objekta i dr.*

Session interfejs obezbeđuje puno metoda. Ovde ćemo pomenuti samo one najvažnije. U dokumentaciji za Hibernate ORM mogu se naći opisi svih metoda.

- ***Transaction beginTraction()*** - počinje jedinica rada i vraća se povezani ***Transaction*** objekat.
- ***void cancelQuery()*** - prekida izvršenje upita
- ***void clear()*** - kompletno čisti sesiju
- ***Connection close()*** - završava sesiju i zatvara JDBC vezu i čisti je.
- ***Criteria createCriteria(Class persistentClass)*** - kreira novi ***Criteria*** objekat, za datu klasu entiteta, ili kreira super klasu klasi entiteta
- ***Serializable getIdentifier(Object object)*** - vraća vrednost indetifikatora datog entiteta u skladu sa vezom sa ovom sesijom.
- ***Query createFilter(Object collection, String queryString)*** - Kreira novi objekatQuery za datu kolekciju i string za filtriranje upita.
- ***Query createQuery(String queryString)*** - kreira nove objekte ***Query*** za dati HQL string upita.
- ***SQLQuery createSQLQuery(String queryString)*** - Kreira novi objekat SQLQuery klase za dati SQL string upita.
- ***void delete(Object object)*** - uklanja trajni objekat iz baze.
- ***void delete(String entityName, Object object)*** - uklanja trajne objekte iz baze

- ***Session get(String entityName, Serializable id)*** - vraća trajni objekat datog imena sa datim identifikatorom, ili ***null*** ako takav objekat ne postoji.
- ***SessionFactory getSessionFactory()*** - daje SessionFactory objekat koji kreira ovu sesiju (Session objekat)..
- ***void refresh(Object object)*** - ponovno čitanje stanja datog objekta iz baze podataka..
- ***Transaction getTransaction()*** - daje Transaction objekat koji je povezan sa ovom sesijom.
- ***boolean isConnected() -*** proverava da li je tekuća sesija povezana
- ***boolean isDirty()*** - da li sesija sadrži neke promene koje se moraju sinhronizovati sa bazom podataka?
- ***boolean isOpen()*** - provera da li je sesija i dalje otvorena
- ***Serializable save(Object object)*** - pre pretvaranja privremenog objekta u trajni, potrebno je da mu se dodeli generisan identifikator
- ***void saveOrUpdate(Object object)*** - ili radi saveObject(Object) update(Object) za dati objekat.
- ***void update(Object object)*** - ažurira trajan objekat sa identifikatorom datog odvojenog objekta
- ***void update(String entityName, Object object)*** - menja trajni objekat sa identifikatorom odvojenog (***detached***) objektu

# HIBERNATE - PERSISTENT KLASA

*Java klase čiji objekti se skladište u bazi podataka se nazivaju trajnim klasama (persistent classes) u Hibernate-u. Kreiraju se primenom POJO pravila.*

Koncept Hibernate ORM se zasniva na uzimanju vrednosti iz atributa Java klase i njihovo smeštanje u tabele baze podataka. Dokument mapiranja pomaže Hibernate-u u određivanju kako da povuče vrednosti iz klasa i da ih mapira sa tabelom i povezanim linkovima.

Java klase čiji objekti se skladište u bazi podataka se nazivaju **trajnim klasama** (persistent classes) u Hibernate-u. Hibernate najbolje radi ako ove klase slede neka prosta pravila, poznata kao Plain Old Java Object (POJO) model programiranja. Ovde se daju sledeća glavna pravila za trajne klase:.

- Sve java klase koje se skladište moraju da imaju unapred definisan konstruktor.
- Sve klase moraju da sadrže **ID** radi lake identifikacije objekata u okviru Hibernate-a i baza podataka. Ovo svojstvo se preslikava direktno u kolonu sa primarnim ključevima u tabeli baze podataka.
- Svi atributi koji se trajno memorišu moraju se deklarisati sa ***private*** i da imaju **getXXX** i **setXXX** metode.
- Centralna funkcija proksija Hibernate-a zavisi od trajne klase koja je ili u stanju ***non-final***, ili je implementirana kao interfejs koji deklariše sve javne metode.
- Klase koje nemaju zahtevana proširenja koje zahteva EJB okvir.

POJO ime se koristi da bi se ukazalo da je dati objekat običan ***Java Object***, ne neki specijalan objekat, a sigurno ne neki ***EntepriseJavaBeans***.

U skladu sa navedenim jednostavnim pravilima, možemo da definišemo jednu POJO klasu na sledeći naičin:

```
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
```

```
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
   public void setSalary( int salary ) {
      this.salary = salary;
   }
}
```

# HIBERNATE - FAJLOVI MAPIRANJA

*Fajl mapiranja kazuje Hibernate-u kako da mapira definisanu klasu ili klase u tabele baze podataka.*

Objektno-relaciono mapiranje se obično definiše u nekom XML dokumentu. Ovaj fajl mapiranja kazuje Hibernate-u kako da mapira definisanu klasu ili klase u tabele baze podataka.

Iako mnogi korisnici Hibernate-a ručno pišu ova XML dokumenta, postoje alati koji generišu ovaj dokument mapiranja.

```
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
```

```
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
   public void setSalary( int salary ) {
      this.salary = salary;
   }
}
```

Za svaki objekat koji želimo da bude trajan, tj. smešten u bazu podataka, odgovara po jedna tabela u bazi, i suprotno. Pretpostavimo da želimo da uskladištimo prethodno prikazane objekte u sledeću RDMS tabelu:

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Na osnovu prikazana dva entiteta, može se kreirati sledeći fajl mapiranja koji pokazuje Hibernate-u kako da mapira prikazanu klasu u tabele baze podataka,

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

<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
   </class>
</hibernate-mapping>
```

# ELEMENTI FAJLA MAPIRANJA

*Dokument mapiranja je XML dokument koji sadrži kao osnovni (root) i element, a on sadrži više elemenata.*

Trebalo bi da memorišete fajl mapiranja u formatu: <classname>.hbm.xml. U našem pokazanom primeru, dokument mapiranja je je **Employee.hbm.xml**. Sada ćemo se pozabaviti sadržajem fajla mapiranja.

Dokument mapiranja je XML dokument koji sadrži <hibernate-mapping> kao osnovni (root) element, a on sadrži više <class> elemenata.

**Elementi <class>** se koriste da bi se definisale specifčnosti mapiranja Java klasa u tabele baze podataka. Naziv Java klase se određuje kada se unose i atributi klase, a naziv tabele u bazi se određuje prilikom upisa atributa u tabelu.

**Element <meta>** je opcioni element i može se koristiti da se kreira opis klase.

**Element <id>** mapira jedinstveni **ID** atribut u klasi u primarne ključeve u tabeli baze. Naziv atributa **id** elementa se odnosi, i ovaj tip tipa a svojstvo u klasi i kolona sa atributima se odnosi na kolonu u bazi podataka. Tip atributa nosi i tip Hibernate mapiranja, i ovi tipovi mapiranja će konvertovati sa Java tipa u u SQL tip podataka.

**Element <generator>** u okviru **<id>** elementa se koristi da automatsku generaciju vrednosti primarnih ključeva. Skup atributa klase class se postavlja na ***native*** (prvobitno) da bi se dopustilo da hibernate pokupi bilo identitet, redosled ili ***hilo*** algoritam radi kreacije primarnog ključa, zavisno od sposobnosti baze podataka koja se koristi.

**Element <property>** se koristi za mapiranje svojstva (atributa) Java klase u kolonu tabele baze. Naziv atributa elementa se odnosi na kolonu tabele baze. Tip atributa nosi i Hibernate tip mapiranja, a ti tipovi mapiranja, konvertuju Java tipove u SQL tipove podataka.

Postoje i drugi atributi i elementi koji se mogu koristiti za mapiranje dokumenta.

# HIBERNATE - TIPOVI MAPIRANJA

*Hibernate tipovi podataka mogu da prevedu Java tipove podataka u SQL tipove podataka i obrnuto.*

Kada pripremate dokument mapiranja Hibernate-a, videli smo da se mapiraju Java tipovi podataka u RDM, tj. SQL tipove podataka. Tipovi koji se deklarišu i koriste u fajlovima mapiranja nisu ni Java tipovi podataka, a ni SQL tipovi podataka. Ovi tipovi podataka su Hibernate tipovi podataka, koji mogu da prevedu Java tipove u SQL tipove podataka i obrnuto.

Ovde dajemo sve osnovne tipove mapiranja, kao i za datum u vreme, za veliki objekat, i za druge tipove mapiranja.

| Mapping type | Java type | ANSI SQL Type |
|---|---|---|
| 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 |

Slika 2.1.4 Tipovi za datum i vreme

| apping type | Java type | ANSI SQL Type |
|---|---|---|
| integer | int or java.lang.Integer | INTEGER |
| long | long or java.lang.Long | BIGINT |
| short | short or java.lang.Short | SMALLINT |
| float | float or java.lang.Float | FLOAT |
| double | double or java.lang.Double | DOUBLE |
| big_decimal | java.math.BigDecimal | NUMERIC |
| character | java.lang.String | CHAR(1) |
| string | java.lang.String | VARCHAR |
| byte | byte or java.lang.Byte | TINYINT |
| boolean | boolean or java.lang.Boolean | BIT |
| yes/no | boolean or java.lang.Boolean | CHAR(1) ('Y' or 'N') |
| true/false | boolean or java.lang.Boolean | CHAR(1) ('T' or |

Slika 2.1.5 Primitivni tipovi

# HIBERNATE - TIPOVI MAPIRANJE (NASTAVAK)

*Binarni i tipovi velikih objekata i tipovi povezani sa JDK*

| Mapping type | Java type | ANSI SQL Type |
|---|---|---|
| binary | byte[] | VARBINARY (or BLOB) |
| text | java.lang.String | CLOB |
| serializable | any Java class that implements java.io.Serializable | VARBINARY (or BLOB) |
| clob | java.sql.Clob | CLOB |
| blob | java.sql.Blob | BLOB |

Slika 2.1.6 Binarni i tipovi velikih objekata

| Mapping type | Java type | ANSI SQL Type |
|---|---|---|
| class | java.lang.Class | VARCHAR |
| locale | java.util.Locale | VARCHAR |
| timezone | java.util.TimeZone | VARCHAR |
| currency | java.util.Currency | VARCHAR |

Slika 2.1.7 JDK tipovi

# MAPIRANJE KOLEKCIJA

*Kolekcije se mogu trajno memorisati primenom java.util.Map, java.util.Set, java.util.SortedMap, java.util.SortedSet, java.util.List, i bilo koj niz trajno memorisanih entiteta ili vrednosti.*

**Mapiranje kolekcija:**

Ako neka klasa ima kolekciju vrednosti za jednu promenljivu, onda se mapiranje ovih vrednosti može izvršiti primenom jednog od interfejsa kolekcija u Javi. Hibernate može trajno memorisati kolekcije primenom ***java.util.Map, java.util.Set, java.util.SortedMp, java.util.SortedSet, java.util.List***, i bilo koji niz trajno memorisanih entiteta ili vrednosti..

| Tip kolekcije | Mapiranje i opis |
|---|---|
| java.util.Set | Mapiranje <set> elemenata i inicijalizano sa java.util.HashSet |
| java.util.SortedSet | Mapiranje <set> elemenata i inicijalizovano sa java.util.TreeSet. Atribut **sort** se može podseiti da primenjuje upoređenje vrednosti ili da ređa elemente na prirodni način. |
| java.util.List | Mapiranje <list> elemenata i inicijalizovano sa java.util.ArrayList. |
| java.util.Collection | Mapiranje <bag> ili <ibag> elemenata i inicijalizovano sa java.util.TArrayList. |
| java.util.Map | Mapiranje <map> elementa i inicijalizovano sa java.util.HashMap. |
| java.util.SortedMap | Mapiranje <map> elementa i inicijalizovano sa java.util.TreeMap. Atribut sort se može podesiti da vrši upoređenje vrednosti (comparator) ili da ređa elemente po prirodnom redosledu (natural oredering) |

Slika 2.1.8 Mapiranje tipova kolekcije

Nizovi su podržani sa Hibernate sa ***<primitive-array>*** za Java tipove primitivnh vrednosti i ***<aray>*** za sve druge tipove, ali se oni retko koriste.

**Mapiranje asocijacija:**

Postoji četiri načina na koji se kardinalnost asocijacije između objekata može izraziti:

| Tipovi mapiranja | Opis |
|---|---|
| Many-to-One | Mapiranje odnosa više-prema-jednom upotrebom Hibrenate-a |
| One-to-One | Mapiranje odnosa jedan-prema-jednom upotrebom Hibrenate-a |
| One-to-Many | Mapiranje odnosa jedan-prema-više upotrebom Hibrenate-a |
| Many-to-Many | Mapiranje odnosa više-prema-više upotrebom Hibrenate-a |

Slika 2.1.9 Mapiranje asocijacija

**Mapiranje komponenata:**

**Komponentna klasa** je klasa koja je vezana za dugu klasu i ni za jednu drugu. Mapiranje kolekcije komponenata se vrši na isti način kao i mapiranje ostalih kolekcija.

# VIDEO: ŠTA JE HIBERNATE?

## *Hibernate Tutorial #1 - Hibernate Overview (8,2 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: HIBERNATE I JDBC

*Hibernate Tutorial #2 - Hibernate and JDBC (1,2 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: INSTALACIJA HIBERNATE

*Hibernate Tutorial #3 - Hibernate Setup Dev Environment Overview (1,24 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: INSTALACIJA ECLIPSE IDE FOR JEEE FOR WINDOWS

*Hibernate Tutorial #4 - Install Eclipse on MS Windows (4,48)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: INSTALACIJA MYSQL FOR MS WINDOWS

*Hibernate Tutorial #5 - Install MySQL Database (4,41 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## 2.1 Pokazni primeri

### PRIMER 1 (1. KORAK) - KREIRANJA POJO OBJEKTA

*POJO (Plain Old Java Object) je Java objekat koji se ne proširuje ili koji ne primenjuje neke specijalizovane klase i interfejse koji se zahtevaju za EJB*

**Kreiranje POJO klasa:**

Prva stvar koja se radi pri razvoja Hibernate aplikacije je formiranje POJO klase ili klasa, zavisno od potreba aplikacije. Uzećemo kao primer klasu **Employee**, sa **getXX i setXX** metodima i napravimo je da bude usaglašena sa zahtevima za ***JavaBeans*** klase.

POJO (Plain Old Java Object) je Java objekat koji se ne proširuje ili koji ne primenjuje neke specijalizovane klase i interfejse koji se zahtevaju za ***EJB*** (***EnterpriseJavaBeans***) radni okvir. Svi normalni Java objekti su POJO.

Kada projektujete klasu koja treba da bude trajna primenom Hibernate ORM, važno je da obezbedite kod koji u saglasnosti sa zahtevima za ***JavaBeans*** klase, kao i da ima jedan atribut koji može da bude indeks, ka na primer, atribut ***id*** kod **Employee** klase.

```
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
   public void setSalary( int salary ) {
      this.salary = salary;
   }
}
```

## PRIMER 2 (2. KORAK) - KREIRANJE TABELA BAZE

*Za svaku trajnu klasu potrebno je kreirati po jednu tabelu u relacionoj bazi podataka.*

Drugi korak je kreiranje tabela u vašoj bazi podataka. Potrebno je da postoji po jedna tabela za svaki objekat koji treba da bude trajan. U našem primeru, imamo samo jednu trajnu klasu Employee, te je potrebno da se kreira jedna tabela.

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

## PRIMER 3 (3. KORAK) - KREIRANJE KONFIGURACIONG FAJLA ZA MAPIRANJE

*Konfiguracioni fajl za mapiranje je XML dokument koji za svoj koren ima <b&amp*

U ovom koraku kreiramo konfiguracioni fajl za mapiranje koji daje informaciju Hibernate-u kako da mapira definisanu klasu (Employee) u tabelu u bazi podataka.

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

<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
   </class>
</hibernate-mapping>
```

Konfiguracion fajl za mapiranje treba da ima sledeći format svog naziva: **<classname>.hbm.xml.** Zato, konfiguracioni fajl za mapiranje klase Emloyee ima naziv: **Employee.hbm.xml**

Konfiguracioni fajl za mapiranje je XML dokument koji za svoj koren ima **<hibernate-mapping>** koji sadrži **<class>** elemente:

- <class> elementi se koriste za definisanje specifičnih mapiranja Java klasa u tabele u bazi. Ime Java klase se određuje imenom atributa klase elementa, a tabela u bazi dobija ime upotrebom imena svog atributa.
- <meta> element je opcioni element i može da se koristi za opisivanje klase.
- <id> element mapira jedinstveni indentifikacioni ID atribut u klasi u primarni ključ tabele u bazi. Ime id atributa odgovara svojstvu (atributu) u klasi i koloni u tabeli. Tip atributa je tip Hibernate mapiranja, i oni vrše konverziju Java tipova u SQL tipove.
- <generator> element se koristi za automatsko generisanje vrednosti primarnog ključa. Ako se ovaj element setuje sa "native" onda Hibernate za primarni ključ uzima identitet, redosled ili hilo algoritam za kreiranje primarnog ključa, zavisno od svojstava odabrnog RDBMS.
- <property> element se koristi za mapiranje svojstva Java klase i kolone u bazi. Ime atributa je isto kao i u klasi i ime kolone tabele. Tip atributa je tip i oni vrše konverziju Java tipova u SQL tipove.

Postoje i drugi atributi i elementi koji se mogu koristiti u dokumentu preslikavanja.

## PRIMER 4 (4. KORAK) - KREIRANJE KLASE APLIKACIJE

*Klasa aplikacije sadrži metod main() neophodan za izvršenje aplikacije. Klasa se koristi za trajno memorisanje objekata u slogove tabele.*

Konačni korak je kreiranje klase aplikacije koja sadrži metod main() za izvršenje aplikacije. Ona se koristi za memorisanje slogova tabele Employee i nad njima onda primenjujemo CRUD operacije.

```
import java.util.List;
import java.util.Date;
import java.util.Iterator;

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

public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      try{
         factory = new Configuration().configure().buildSessionFactory();
      }catch (Throwable ex) {
```

```
         System.err.println("Failed to create sessionFactory object." + ex);
         throw new ExceptionInInitializerError(ex);
      }
      ManageEmployee ME = new ManageEmployee();

      /* Add few employee records in database */
      Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);
      Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
      Integer empID3 = ME.addEmployee("John", "Paul", 10000);

      /* List down all the employees */
      ME.listEmployees();

      /* Update employee's records */
      ME.updateEmployee(empID1, 5000);

      /* Delete an employee from the database */
      ME.deleteEmployee(empID2);

      /* List down new list of the employees */
      ME.listEmployees();
   }
   /* Method to CREATE an employee in the database */
   public Integer addEmployee(String fname, String lname, int salary){
      Session session = factory.openSession();
      Transaction tx = null;
      Integer employeeID = null;
      try{
         tx = session.beginTransaction();
         Employee employee = new Employee(fname, lname, salary);
         employeeID = (Integer) session.save(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
      return employeeID;
   }
   /* Method to  READ all the employees */
   public void listEmployees( ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         List employees = session.createQuery("FROM Employee").list();
         for (Iterator iterator =
                           employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print("  Last Name: " + employee.getLastName());
            System.out.println("  Salary: " + employee.getSalary());
```

```
         }
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to UPDATE salary for an employee */
   public void updateEmployee(Integer EmployeeID, int salary ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                    (Employee)session.get(Employee.class, EmployeeID);
         employee.setSalary( salary );
         session.update(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to DELETE an employee from the records */
   public void deleteEmployee(Integer EmployeeID){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                   (Employee)session.get(Employee.class, EmployeeID);
         session.delete(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
}
```

## PRIMER 5 (5. KORAK) - KOMPILACIJA I IZVRŠENJE

*Kompilacija i izvršenje su poslednje aktivnosti za dobijanje rezultata izvršenje neke aplikacije.*

Proverite da li ste dobro podesili PATH i CLASSPATH na odgovarajući način pre nego što uradite kompiliranje (prevođenje izvrong koda u izvršni) i izvršenje programa.

1. Kreirajte hibernate.cfg.xml konfiguracioni fajl
2. Kreirajte Employee.hbm.xml fajl za preslikavanje
3. Kreirajte Employee.java fajl sa izvornim programskim kodom i izvršite kompilaciju..
4. Kreirajte ManageEmployee-java fajl sa izvornim programskim kodom i izvršite kompilaciju (prevođenje).
5. Izvršite ManageEmployee izvršni (binarni) kod

Dobićete sledeći rezultat

```
$java ManageEmployee .....
Ovde se prikazuju različite poruke.....

First Name: Zara Last Name: Ali Salary: 1000
First Name: Daisy Last Name: Das Salary: 5000
First Name: John Last Name: Paul Salary: 10000
First Name: Zara Last Name: Ali Salary: 5000
 First Name: John Last Name: Paul Salary: 10000
```

Slika 2.2.1 Rezultat

## VIDEO: UNOŠENJE POČETNIH PODATAKA U BAZI

*Hibernate Tutorial #6 - Setup Sample Data (7,25 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO: POSTAVLJANJE HIBERNATE U ECLIPSE IDE

*Hibernate Tutorial #7 - Setup Hibernate in Eclipse (10,30 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO: TESTIRANJE JDBC VEZE

*Hibernate Tutorial #8 - Test JDBC Connection (6,19 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## VIDEO: PROCES RAZVOJA HIBERNATE APLIKACIJE - UVOD

*Hibernate Tutorial #9 - Hibernate Dev Process (0,44 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

### VIDEO: KONFIGURISANJE HIBERNATE ORM

*Hibernate Tutorial #10 - Hibernate Configuration*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## 2.2 Zadaci za samostalni rad

### ZADACI ZA SAMOSTALNI RAD STUDENTA

*Cilj je provežbavanje naučenog*

**Zadatak 2:**

Za sistem po izboru (apoteka, biblioteka, fakultet ...) započeti kreiranje CRUD aplikacije korišćenjem Hibernate ORM-a. Prateći korake primera od 1 do 5 uradite kreiranje POJO klasa i podesite konfiguraciju projekta.

# Poglavlje 3

# Hibernate Annotations (napomene)

## HIBERNATE NAPOMENE

*Hibernte Annotations je najnoviji način za definisanje mapiranja objekata u tabele i obrnuto, bez upotrebe XML fajla.*

Hibernte Annotations je najnoviji način za definisanje mapiranja objekata u tabele i obrnuto, bez upotrebe XML fajla. Možete koristiti Hibernate Annotations kao dopunu ili kao zamenu XML mapiranja meta podataka.

Hibernate Annotations tj. korišćenje napomena, je moćan način obezbeđivanja meta podataka za mapiranje objekata u relacione tabele. Svi meta podaci se stavljaju zajedno u POJO java fajl zajedno sa programskim kodom, te korisnik razume strukturu tabela i POJO simultano za vreme razvoja.

Ako želita da razvijete vašu prenosivu aplikaciju za druge *EJB 3 ORM* aplikacije, vi morate da koristite napomene, tj. Hibernate Annotations da bi predstavili informaciju o mapiranju. Time, dobijate i veću fleksibilnost nego da ste koristili XML definisanje potrebnog mapiranja.

**Podešavanje radnog okruženja za Hibernate Annotations:**

Morate koristiti *JDK 5.0* da bi koristili podršku napomenama koje Java obezbeđuje. Morate i da instalirate *Hibernat 3.x annotations* distribucioni paket, a koji je raspoloživ sa Hibernate izvora. Znači, treba da preuzmete Hibernate Annotation i da koprate ***hibernate-annotations.jar, hibernate-comons-annotations.jar*** na Vaš CLASSPATH. .

**Primer:**

Pretpostavimo da vaša baza kooristi sledeću tabelu **EMLPOYEE** za smeštaj podataka o zaposlenima.

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Sledeći kod pokazuje kako bi koristili Javva Annotations za mapiranje objekata u EMPLYEE tabelu:

```
import javax.persistence.*;

@Entity
@Table(name = "EMPLOYEE")
public class Employee {
   @Id @GeneratedValue
   @Column(name = "id")
   private int id;

   @Column(name = "first_name")
   private String firstName;

   @Column(name = "last_name")
   private String lastName;

   @Column(name = "salary")
   private int salary;

   public Employee() {}
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
   public void setSalary( int salary ) {
      this.salary = salary;
   }
}
```

# NAJČEŠĆE KORIŠĆENE HIBERNATE NAPOMENE

*Napomene: @Entity, @Table, @Id i @GeneratedValue daju odgovarajuće instrukcije mapiranja programu tokom njegovog izvršavanja.*

Hibernate nalazi ***@Id***anotaciju i predpostavlja da treba onda da pristupi svojstvima nekog objekta direktno za vreme izvršenja programa. Ako ste stavili ***@Id*** napomenu u **getId()** metod vi ste omogućili pristup svojstvima objekta preko **geter** i **seter** metoda. Sve ostale napomene se postavljaju ili u polja (atribute) objekata ili u geter metode, sledeći istu strategiju. Ova sekcija objašnjava upotrebu napomena za slučaj klase **Employee** korišćenu u prethodnom primeru.

***@Entity*** **napomena**

***EJB 3*** standardne napomene (annotations) se nalaze u ***javax.persistence*** paketu. Koristimo ***@Entity annotation*** u klasi **Employee** koja čini ovu klasu kao ***entitity bean***, te ne sme da ima argumente u konstruktoru koji su vidljivi više od nivo ***protected.*** .

***@Table*** napomena:

***@Table*** napomena dozvoljava vam da specificirate detalje tabele koja će se koristiti za trajano memorisanje (smeštaj) u bazi podataka. @Table anotacija obezbeđuje četiri atributa koji vam omogućuju da promenite ime tabele, njen katalog i njenu šemu i da time nametnete jedinstvena ograničenja kolonama tabele.. Do sada smo koristili naziv tabele EMPLOYEE.

***@Id*** i ***@GeneratedValue*** napomene:

Svaka klasa tipa ***entity bean*** imaće primarni ključ koji u klasi označite napomenom ***@Id.*** Primarni ključ može imati svoje posebno polje (atribut) u klasi ili može da bude kombinovan sa više polja, zavisno od vaše strukture tabela. Napomena, @Id će automatski odrediti primarni ključ koji najviše odgovara generisanoj strategiji mapiranja. Međutim, vi to možete promeniti primenom napomene ***@GeneratedValue*** koja ima dva parametara: ***strategy*** i ***generator.*** Ostavljajući da Hibernate odredi koji će generisan tip da se koristi čini vaš programski kod prenosivim za slučaj korišćenja različitih baza podataka.

***@Column*** **napomena:**

***@Column*** napomena se koristi za specificiranje polja ili svojstva klase koje se mapira u kolonu tabele. Najčešće korišćeni, atributi:

- ***name*** ime kolone koja se ovde navodi.
- ***length*** atribut određuje veličinu kolone pri mapiranju tipa String.
- ***nullable*** atribut dozvoljava da kolona može da bude obeležena sa NOT NULL pri generisanju tabele..
- ***unique*** atribut dozvoljava koloni da koristi samo jedinstvene vrednosti (tipa ***unique***)

# VIDEO: HIBERNATE ANNOTATIONS - 1

## *Hibernate Tutorial #11 - Hibernate Annotations - Part 1 (6,51 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: HIBERNATE ANNOTATIONS - 2

*Hibernate Tutorial #12 - Hibernate Annotations - Part 2 luv2code (7,28 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: SKLADIŠĆENJE JAVA OBJEKTA - 1

*Hibernate Tutorial #13 - Save a Java Object - Part 1 (6,11 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: SKLADIŠĆENJE JAVA OBJEKTA - 2

*Hibernate Tutorial #14 - Save a Java Object - Part 2 (9,11 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: PRIMARNI KLJUČEVI - 1

*Hibernate Tutorial #15 - Primary Keys - Part 1 luv2code (6,11 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: PRIMARNI KLJUČEVI - 2

*Hibernate Tutorial #16 - Primary Keys - Part 2 (7,11 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: PRIMARNI KLJUČEVI - 3

*Hibernate Tutorial #17 - Primary Keys - Part 3 (3,08 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: ČITANJE OBJEKATA PRIMENOM PRIMARNOG KLJUČA

*Hibernate Tutorial #18 - Reading Objects with Primary Key (9,52)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## 3.1 Pokazni primer

### PRIMER 6 - PRIMER PRIMENE HIBERNATE NAPOMENA - KLASA/ TABELA EMLOYEE

*Aplikaciona klasa ManageEmployee sadrži metod main() koji omogućava izvršavanje aplikacije, hibernate.cfg.xml vrši konfigurisanje baze podataka radi preslikavanja klase u tabelu.*

**Kreiranje aplikacione klase:**

Aplikaciona klasa sadrži metod main() koji omogućava izvršavanje aplikacije. Daćemo ovde primer aplikacije koja realizuje nekolio CRUD operacija sa nekoliko slogova tabele EMPLOYEE, tj. nekoliko objekata klase Employee.

```
import java.util.List;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.cfg.AnnotationConfiguration;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      try{
         factory = new AnnotationConfiguration().
                   configure().
                   //addPackage("com.xyz") //add package if used.
                   addAnnotatedClass(Employee.class).
```

```
                buildSessionFactory();
    }catch (Throwable ex) {
       System.err.println("Failed to create sessionFactory object." + ex);
       throw new ExceptionInInitializerError(ex);
    }
    ManageEmployee ME = new ManageEmployee();

    /* Add few employee records in database */
    Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);
    Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
    Integer empID3 = ME.addEmployee("John", "Paul", 10000);

    /* List down all the employees */
    ME.listEmployees();

    /* Update employee's records */
    ME.updateEmployee(empID1, 5000);

    /* Delete an employee from the database */
    ME.deleteEmployee(empID2);

    /* List down new list of the employees */
    ME.listEmployees();
}
/* Method to CREATE an employee in the database */
public Integer addEmployee(String fname, String lname, int salary){
    Session session = factory.openSession();
    Transaction tx = null;
    Integer employeeID = null;
    try{
       tx = session.beginTransaction();
       Employee employee = new Employee();
       employee.setFirstName(fname);
       employee.setLastName(lname);
       employee.setSalary(salary);
       employeeID = (Integer) session.save(employee);
       tx.commit();
    }catch (HibernateException e) {
       if (tx!=null) tx.rollback();
       e.printStackTrace();
    }finally {
       session.close();
    }
    return employeeID;
}
/* Method to  READ all the employees */
public void listEmployees( ){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
       tx = session.beginTransaction();
       List employees = session.createQuery("FROM Employee").list();
       for (Iterator iterator =
```

```
                           employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print("  Last Name: " + employee.getLastName());
            System.out.println("  Salary: " + employee.getSalary());
         }
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to UPDATE salary for an employee */
   public void updateEmployee(Integer EmployeeID, int salary ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                    (Employee)session.get(Employee.class, EmployeeID);
         employee.setSalary( salary );
         session.update(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to DELETE an employee from the records */
   public void deleteEmployee(Integer EmployeeID){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                   (Employee)session.get(Employee.class, EmployeeID);
         session.delete(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
}
```

**Konfigurisanje baze podataka:**

Sada treba kreirati hibernate.cfg.xml konfiguracioni fajl radi definisanja pojedinih parametara baze podataka.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>

   <!-- Assume students is the database name -->
   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/test
   </property>
   <property name="hibernate.connection.username">
      root
   </property>
   <property name="hibernate.connection.password">
      cohondob
   </property>

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

## PRIMER 6 - PRIMER PRIMENE HIBERNATE NAPOMENA - KLASA/ TABELA EMLOYEE (NASTAVAK)

*Pre izvršenja programa, mora da se podese vrednosti za PATH i CLASSPATH i kreirati i kompilirati fajlove sa izvornim kodomi mapiranja.*

**Kompilacija i izvršenje programa:**

Prvo, treba da podesite vrednosti za PATH i CLASSPATH i onda da sprovedete sledeće korake:

- Izbrišite **Employee.hbm.xml** fajla za mapiranje sa puta (PATH).
- Kreirajte **Employee.java** izvorni fajl (kao što je urađeno u prethondpj sekciji) i izvršiti kompilaciju.
- Kreirajte **ManageEmployee.java** izvorni fajl (dat u prethodnoj sekciji) i izvršite kompilaciju.
- Izvršite binarni fajl **ManageEmployee**

Dole se daje dobijeni rezultat:e.

```
$java ManageEmployee .....
Ovde se prikazuju različite log poruke.....

First Name: Zara Last Name: Ali Salary: 1000
First Name: Daisy Last Name: Das Salary: 5000
First Name: John Last Name: Paul Salary: 10000
First Name: Zara Last Name: Ali Salary: 5000
First Name: John Last Name: Paul Salary: 10000
```

Slika 3.1.1 Sadržaj dobijene tabele EMPLOYEE

## 3.2 Zadaci za samostalni rad

### ZADACI ZA SAMOSTALNI RAD STUDENTA

*Cilj je provera stečenog znanja*

**Zadatak 3:**

Po uzoru na primer 6 prepraviti kod zadatka 2 tako da koriste Hibernate anotacije ne xml preslikavanje.

# Poglavlje 4

# Hibernate Quary Language - HQL jezik za upite

## PRIMER 7 - INSTRUKCIJE HQL - FROM I AS

*Instrukcija FROM prebacuje trajne objekte u memoriju računara, a AS se koristi za dodeljivanje nadimaka, a radi skraćivanja naziva klasa i HQL upita.*

***Hibernate Query Language (HQL)*** je objektno-orijentisan jezik za rad sa bazom podataka u okviru Hibernate ORM. HQL radi sa trajnim objektima i njihovim svojstvima. ***HQL*** upiti se prevode u uobičajene ***SQL*** upite (od strane Hibernate-a) koji izvršavaju potrebne akcije na bazi podataka.

Vi možete i direktno da koristite ***SQL*** upita i u okviru Hibernate-a upotrebmo ***Native SQL***, ali je prporučljivo da koristite ***HQL*** uvek kada možete, da bi izbegli probleme sa prenosivošću aplikacije (sa jedne na drugu bazu podataka), ***HQL*** takođe nudi i neke prednosti, kao što su strategije generisanja i keširanje

Ključne reči kao što su SELECT , FROM, WHERE i dr. u ***HQL*** ne zavise od veličine slova, ali svojstva, kao što su imena tabela i kolona zavise od veličine slova.

**FROM insrukcija:**

Upotrebljavate FROM instrukciju, ako želite da prebacite sve trajne objekte u memoriju računara. Možete koristiti sledeću jednostavnu sintaksu:

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
List results = query.list();
```

Ako želite da koristite sve objekte jedne klase, onda možete dati sledeće instrukcije:

```
String hql = "FROM Employee AS E";
Query query = session.createQuery(hql);
List results = query.list();
```

**AS instrukcija:**

Instrukcija AS se koristi za dodeljivanje nadimaka (skraćenica), tj. kraćih imena naziva klasa u većim ***HQL*** upitima. To se koristi najviše da bi se izbeglo korišćenje dugih naziva klasa i da bi

se skratili upiti. Na primer, u prethodnom primeru, korišćenje sinonima ili skraćenica, dovodi do sledećih ***HQL*** upita:

```
String hql = "FROM Employee AS E";
Query query = session.createQuery(hql);
List results = query.list();
```

Ključna reč se može izostaviti i možete da specificirate samo nadimak (skraćeni naziv) direktno na mestu imena klase, kao što se ovde vidi:

```
String hql = "FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

# PRIMER 7 - INSTRUKCIJE HQL - SELECT, WHERE, ORDERED BY I GROUPED BY

*Instrukcije SELECT i WHERE sužavaju izbor svojstava objekata koji se preuzimaju iz baze, ORDERED BY vrši sortiranje rezultata, a GROUP BY grupiše ih po nekoj vrednosti svojstava.*

SELECT **instrukcije**: Instrukcija SELECT obezbeđuje veću kontrolu skupa objekata koji se dobije kao rezultat upita. Ako želite da dobijete samo neka svojstva objekta, a ne sva, onda koristite ***SELECT*** instrukciju. Evo primera sa SELECT instrukciom kojom se dobija svojstvo firstName objekta **Employee**:

```
String hql = "SELECT E.firstName FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

Može se uočiti da je dobijeno svojstvo objekta **Employee.firstName** objekta **Employee** , a ne polje u **EMPLOYEE** tabeli.

WHERE **instrukcija**: Ako želite da suzite set specifičnih objekata u rezultatu upita, a koji dolaze iz trajnog skladišta, onda koristite WHERE instrukciju. Evo jednostavne sintakse upotrebe WHERE instrukcije:

```
String hql = "FROM Employee E WHERE E.id = 10";
Query query = session.createQuery(hql);
List results = query.list();
```

ORDER BY instrukcija: U cilju sortiranja rezultata HQL upita, treba da koristiti istrukciju ***ORDERED BY***. Sortiranje možete izvršite prema bilo kom svojstvu objekta po rastućem redosledu (ASC) ili opadajućem redosledu (DESC). Evo primera:

```
String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";
Query query = session.createQuery(hql);
List results = query.list();
```

Ako želite da izvršite sortiranje po više svojstava, možete dodati dodatna svojstva na kraju instrukcije, koristeći kao separator zapete:

```
String hql = "FROM Employee E WHERE E.id > 10 " +
             "ORDER BY E.firstName DESC, E.salary DESC ";
Query query = session.createQuery(hql);
List results = query.list();
```

GROUP BY **instrukcije:** Ova instrukcija dovodi informacije iz baze podataka i njihovo grupisanje u odnosu na vrednost atributa i dodavat i zbirnu vrednost svakog svojstva. Evo primera jednostavne sintakse primene instrukcije GROUP BY:

```
String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E " +
             "GROUP BY E.firstName";
Query query = session.createQuery(hql);
List results = query.list();
```

# PRIMER 7 - INSTRUKCIJE HQL - UPDATE, DELETE I INSERT

*HQL instrukcije UPDATE, DELETE i INSERT promenu vrednosti atributa, brisanje objekta ili ubacivanje novog jednog ili više objekata.*

**Upotreba imenovanih parametara:**

Hibernate podržava imenovane parametre u svojim HQL upitima. To čini pisanje HQL upita, koje obavlja korisnik, lakim i nemate potrebu da se branite od "SQL injection" napada. Sledi primer koda/sintakse upotrebe imenovanih parametara:

```
String hql = "FROM Employee E WHERE E.id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id",10);
List results = query.list();
```

UPDATE **instrukcije**

Metod ***executeUpdate()*** izvršava HQL ***UPDATE*** ili ***DELETE*** iskaze. ***UPDATE*** iskaz vrši promenu vrednosti svojstava (atributa) jednog ili više objekata. Sledeći primer daje primenu ***UPDATE*** sintakse.

```
String hql = "UPDATE Employee set salary = :salary "  +
             "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("salary", 1000);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

DELETE **instrukcija:**

DELETE instrukcija briše jedan ili više objekata. Evo jednog primera:

```
String hql = "DELETE FROM Employee "  +
             "WHERE id = :employee_id";
Query query = session.createQuery(hql);
query.setParameter("employee_id", 10);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

INSERT **instrukcija:**

HQL podržava ***INSERT INTO*** instrukciju samo u slučaju kada slog može da se ubaci iz jednog objkekta u drugi objekat. Evo jednog primera:

```
String hql = "INSERT INTO Employee(firstName, lastName, salary)"  +
             "SELECT firstName, lastName, salary FROM old_employee";
Query query = session.createQuery(hql);
int result = query.executeUpdate();
System.out.println("Rows affected: " + result);
```

# PRIMER 7 - INSTRUKCIJE HQL - AGREGATNI METODI I PAGINACIJA

*HQL agregatni metodi omogućavaju selektivni izbor objekata i analizu vrednosti svojstava objekata. HQL metodi za paginaciju, određuju veličinu jedne strane izveštaja sa rezultatioma.*

**Agregatni metodi:**

HQL podržava opseg agregatnioh metoda, slično kao što do radi SQL. Evo primera:

| RB | Opisi | Opisi |
|---|---|---|
| 1 | avg(naziv svojstva) | Srednja vrednost svojstava |
| 2 | count(naziv svojstva ili *) | Broj ponavljanja svojstva u rezultatu |
| 3 | max(naziv svojstva) | Maksimalna vrednost svih vrednosti svojstva. |
| 4 | min(naziv svojstva) | Minimalna vrednost svih vrednosti svojstva. |
| 5 | sum(naziv svojstva) | Zbir svih vrednosti svojstva |

Slika 4.1.1 Agregatni metodi HQL

Posebna ključna reč ***dinstict*** se odnosi samo na vrednosti jednog reda. Sledeći upit vraća samo jednu vrednost:

```
String hql = "SELECT count(distinct E.firstName) FROM Employee E";
Query query = session.createQuery(hql);
List results = query.list();
```

**Paginacija**

Koriste se dva metoda HQL za paginaciju, tj. za određivanje veličine jedne strance prikaza rezultata HQL upita. Ukupan broj strana izveštaja onda zavisi od količine izlaznih rezultata.

1 ***Query setFirstResult(int startPosition)***

Metod određuje uneti ceo broj da predstavlja prvi red u vašem prikazu rezultata, polazeći od reda 0.

2 ***Query setMaxResults(int maxResult)***

Ovaj metod saopštava Hiberante-u da preuzme određen broj ***maxResults*** objekata.

Upotrebom oba gornja metoda zajedno, možemo da napravimo ostraničenu komponentu na našem vebu ili u Swing aplikaciji. Evo jednog primera koji možete da se proširite tako da obuhvatii 10 redova na jednoj stranici..

```
String hql = "FROM Employee";
Query query = session.createQuery(hql);
query.setFirstResult(1);
query.setMaxResults(10);
List results = query.list();
```

# 4.1 Zadaci za samostalni rad

## ZADATAK 4

*Cilj zadataka za samostalan rad je da student primeni stečeno znanje*

Kreirati bazu podataka koja sadrži tabelu student i koja od atributa ima id, ime, prezime, brojIndeksa, prosek.

Nakon toga kreirati i konfigurisati Hiberante projekat koji će biti povezan sa kreiranom bazom podataka. Potrebno je da se izgeneriše entitet klasa Student kao što je urađeno na vežbama.

Zatim kreirati klasu koja će realizovati CRUD operacije nad bazom korišćenjem HQL-a . Pored CRUD operacija dodati i metodu koja pronalazi sve studente koji imaju prosek iznad 9.

Nakon toga je potrebno kreirati JavaFX formu za dodavanje novih studenata kao i prozor u okviru kog će se prikazati svi Studenti iz baze sa mogućnošću brisanja studenata iz baze kao i ažuriranja podataka o studentima.

Kreirati posebno dugme koje će pretražiti samo studente koji imaju prosek veći od 9 i prikazati ih u prozoru za prikaz studenata.

# Poglavlje 5

# Kriterijumi za izbor objekata u HQL upitu

## OBJEKAT CRITERIA

*Objekat Criteria vam omogućava da definišete kriterijume za programirani odabir objekata primenom pravila filtriranja i logičkih uslova.*

Postoji više načina da manipulište sa objektima i odredite koji će objekti da se trajno uskladište iz vaše aplikacije, ili obrnuto, koji će se objekti iz skladišta (trajne memorije računara) da preuzmu i prebace u radnu memoriju aplikacije. Jedan od ovih načina je korišćenje metoda datih u Crteria API Hibernate-a. Ovaj interfejs vam omogućava da definišete kriterijume za programirani odabir objekata primenom pravila filtriranja i logičkih uslova.

Hibernate Session intefejs obezbeđuje metod createCriteria koji kreira objekat Criteria koji vraća objekte trajne klase kada u vašoj aplikaciji izvršavate upit sa ovim kriterijumima.
Sledeći jednostavan primer prikazuje upit sa postavljenim kriterijumima za izbor objekata iz klase **Employee**.

```
Criteria cr = session.createCriteria(Employee.class);
List  results  = cr.list();
```

**Ograničenja objekta Criteria:**

Možete koristiti metod ***add()*** objekta Criteria da bi dodali dodatne kriterijume u vaš upit. U sledećem primeru dodat je dodatni kriterijum da se dobiju samo slogovi (iz baze) u kojima je plata jednaka 2000.

```
Criteria cr  = session.createCriteria( Employee.class );
cr.add( Restrictions.eq("salary",2000) );
List results  =  cr.list();
```

Evo nekoliko primera koji daju različita scenarija koji se mogu koristiti u nekom upitu.

```
Criteria cr = session.createCriteria(Employee.class);

// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));
```

```
// To get records having salary less than 2000
cr.add(Restrictions.lt("salary", 2000));

// To get records having fistName starting with zara
cr.add(Restrictions.like("firstName", "zara%"));

// Case sensitive form of the above restriction.
cr.add(Restrictions.ilike("firstName", "zara%"));

// To get records having salary in between 1000 and 2000
cr.add(Restrictions.between("salary", 1000, 2000));

// To check if the given property is null
cr.add(Restrictions.isNull("salary"));

// To check if the given property is not null
cr.add(Restrictions.isNotNull("salary"));

// To check if the given property is empty
cr.add(Restrictions.isEmpty("salary"));

// To check if the given property is not empty
cr.add(Restrictions.isNotEmpty("salary"));
```

# CRITERIA API - PAGINACIJA I SORTIRANJE

*Primenom metoda setFirstResult i setMaxResult može se definisati paginacija rezultata, a primenom klase Order, može se izvršiti sortiranje dobijenih rezultata.*

Možete da kreirate ***AND*** ili ***OR*** uslove upotrebom ***LogicalExpresion*** ograničenja, kao što je prikazano u sledećem primeru:

```
Criteria cr = session.createCriteria(Employee.class);

Criterion salary = Restrictions.gt("salary", 2000);
Criterion name = Restrictions.ilike("firstNname","zara%");

// To get records matching with OR condistions
LogicalExpression orExp = Restrictions.or(salary, name);
cr.add( orExp );


// To get records matching with AND condistions
LogicalExpression andExp = Restrictions.and(salary, name);
cr.add( andExp );
```

```
List results = cr.list();
```

**Paginacija upotrebom objekta *Criteria***

Objekat Criteria nudi dva metoda za paginaciju.

1 ***public Criteria setFirstResult(int firstResult)***

Metod uzima neki ceo broj za označavanje prvog sloga u vašem setu rezultata, polazeći od reda 0.
2 ***public Criteria setMaxResults(int maxResults)***
Ovaj metod kaže Hibernate-u da pozovi određen broj ***maxResults*** objekata

Upotrebom navedena dva metoda, može se napraviti komponenta za paginaciju za veb ili Swing aplikaciju. U sledećem primeru možete proširiti na 10 redova po strani:

```
Criteria cr = session.createCriteria(Employee.class);
cr.setFirstResult(1);
cr.setMaxResults(10);
List results = cr.list();
```

**Sorttirnje rezultata:**

Criteria API obezbeđuje klasu ***org.hibernate.criterion.Order*** koja sortira set rezultata po rastućem ili opadajućem redosledu vrednosti nekog svojstva objekta. Evo primera korišćenja klase Order:

```
Criteria cr = session.createCriteria(Employee.class);
// To get records having salary more than 2000
cr.add(Restrictions.gt("salary", 2000));

// To sort records in descening order
crit.addOrder(Order.desc("salary"));

// To sort records in ascending order
crit.addOrder(Order.asc("salary"));

List results = cr.list();
```

# CRITERIA API - KLASA PROJECTIONS

*Criteria API obezbeđuje klasu org.hibernate.criterion.Projections koja obezbeđuje metode za određivanje srednje vrednosti, maksimalne i minimalne vrednosti dobijenih rezultata*

**Projekcije i agregacije:**

Criteria API obezbeđuje klasu ***org.hibernate.criterion.Projections*** koja obezbeđuje metode za određivanje srednje vrednosti, maksimalne i minimalne vrednosti dobijenih rezultata. Klasa ***Projections*** je slična klasi ***Restrictions*** jer obezbeđuje nekoliko statičkih metoda tipa ***Factory*** tj. koji prave objekte klase ***Projections***. Evo nekoliko primera sa nekoliko scenarija po svakom zahtevu:

```
Criteria cr = session.createCriteria(Employee.class);

// To get total row count.
cr.setProjection(Projections.rowCount());

// To get average of a property.
cr.setProjection(Projections.avg("salary"));

// To get distinct count of a property.
cr.setProjection(Projections.countDistinct("firstName"));

// To get maximum of a property.
cr.setProjection(Projections.max("salary"));

// To get minimum of a property.
cr.setProjection(Projections.min("salary"));

// To get sum of a property.
cr.setProjection(Projections.sum("salary"));
```

# VIDEO: POZIVANJE OBJEKATA PRIMENO UPITA HIBERNATE UPITA HQL - 1

*Hibernate Tutorial #19 - Querying Objects with HQL - Part 1 (3,31 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: POZIVANJE OBJEKATA PRIMENO UPITA HIBERNATE UPITA HQL - 2

*Hibernate Tutorial #20 - Querying Objects with HQL - Part 2 (13,53 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: PROMENA OBJEKTA U TRAJNOM SKLADIŠTU SA HQL - 1

*Hibernate Tutorial #21 - Update Objects - Part 1 (3,37 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: PROMENA OBJEKTA U TRAJNOM SKLADIŠTU SA HQL - 2

*Hibernate Tutorial #22 - Update Objects - Part 2 (7,57 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: BRISANJE OBJEKATA U TRAJNOM SKLADIŠTU SA HQL - 1

*Hibernate Tutorial 23 - Delete Objects - Part 1 (2,45 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

# VIDEO: BRISANJE OBJEKATA U TRAJNOM SKLADIŠTU SA HQL - 2

*Hibernate Tutorial #24 - Delete Objects - Part 2 (6,33 minuta)*

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## 5.1 Pokazni primeri

### PRIMER 8 - PRIMER PRIMENE CRITERIA API

*Prvo se definiše POJO klasa Emloyee, a zatim tabela baze EMLPOYEE i XML fajl za mapiranje objekata Employee u slogove tabele EMPLOYEE.*

Pretpostavimo sledeću POJO klasu:

```
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
   public void setSalary( int salary ) {
      this.salary = salary;
   }
}
```

Sada, kreirajmo tabelu **EMPLOYEE** koja skladišti **Employee** objekte:

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Kreirajmo fajl mapiranja:

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

<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
   </class>
</hibernate-mapping>
```

## PRIMER 8 - PRIMER PRIMENE CRITERIA API - KLASA APLIKACIJE MANAGEEMPLOYEE

### *Klasa aplikacije ManageEmployee sadrži metod main() koji obezbeđuje HQL upite sa uslovima definisani primenom Criteria API*

Sada, kreiramo klasu aplikacije sa metodomom ***main(),*** koji će koristiti HQL upite sa uslovima definisani primenom Criteria API:

```
import java.util.List;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.Criteria;
import org.hibernate.criterion.Restrictions;
import org.hibernate.criterion.Projections;
import org.hibernate.cfg.Configuration;

public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      try{
         factory = new Configuration().configure().buildSessionFactory();
      }catch (Throwable ex) {
         System.err.println("Failed to create sessionFactory object." + ex);
         throw new ExceptionInInitializerError(ex);
```

```
      }
      ManageEmployee ME = new ManageEmployee();

      /* Add few employee records in database */
      Integer empID1 = ME.addEmployee("Zara", "Ali", 2000);
      Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
      Integer empID3 = ME.addEmployee("John", "Paul", 5000);
      Integer empID4 = ME.addEmployee("Mohd", "Yasee", 3000);

      /* List down all the employees */
      ME.listEmployees();

      /* Print Total employee's count */
      ME.countEmployee();

      /* Print Toatl salary */
      ME.totalSalary();
   }
   /* Method to CREATE an employee in the database */
   public Integer addEmployee(String fname, String lname, int salary){
      Session session = factory.openSession();
      Transaction tx = null;
      Integer employeeID = null;
      try{
         tx = session.beginTransaction();
         Employee employee = new Employee(fname, lname, salary);
         employeeID = (Integer) session.save(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
      return employeeID;
   }

   /* Method to  READ all the employees having salary more than 2000 */
   public void listEmployees( ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Criteria cr = session.createCriteria(Employee.class);
         // Add restriction.
         cr.add(Restrictions.gt("salary", 2000));
         List employees = cr.list();

         for (Iterator iterator =
                           employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print("  Last Name: " + employee.getLastName());
```

```
            System.out.println("  Salary: " + employee.getSalary());
         }
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to print total number of records */
   public void countEmployee(){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Criteria cr = session.createCriteria(Employee.class);

         // To get total row count.
         cr.setProjection(Projections.rowCount());
         List rowCount = cr.list();

         System.out.println("Total Coint: " + rowCount.get(0) );
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
  /* Method to print sum of salaries */
   public void totalSalary(){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Criteria cr = session.createCriteria(Employee.class);

         // To get total salary.
         cr.setProjection(Projections.sum("salary"));
         List totalSalary = cr.list();

         System.out.println("Total Salary: " + totalSalary.get(0) );
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
```

```
}
```

## PRIMER 8 - PRIMENA CRITERIA API - KOMPILACIJA I IZVRŠENJE

*Kriran fajl sa izvršnim kodom ManageEmployee.java se prevodi (kompilira) i dobijeni izvršni (binarni) program se izvršava, Dobijena je tabela EMPLOYEE sa trajnim objektima Employee.*

**Kompilacija i izvršenje:**

Prvo, treba da podesimo PATH i CLASSPATH parametre pre nego što pristupimo proceduri kompilacije i izvršenju programa. Zatim, primenjujemo preporučene korake kompilacije i izvršenja programa.

1. Kreiranje ***hibernate.cfg.xml*** fajla konfiguracije (urađeno ranije)
2. Kreiranje ***Employee.hbm.xml*** fajla za mapiranje (kao što je već prikazano)
3. Kreiranje ***ManageEmployee.java*** fajla sa izvornim kodom (kao što je već urađeno) i njena kompilacija
4. Izvršenje **ManageEmployee** izvršnog (binarnog) programa .

Trebalo bi da dobijete sledeći razultat.

```
$java ManageEmployee
 .....prika različitih log poruka.....
First Name: Daisy    Last Name: Das Salary: 5000
First Name: John     Last Name: Paul Salary: 5000
First Name: Mohd     Last Name: Yasee Salary: 3000
Total Count: 4
Total Salary: 15000
```

Slika 5.1.1 Prikaz dobijenog rezultata

Ako izvršite proveru tabele EMPLOYEE sa trajno memorisanim slogovima - objektima Employee, trebalo bi da dobijete sledeću tabelu:

```
mysql> select * from EMPLOYEE;
+----+--------------+---------------+--------+
| id | first_name | last_name | salary |
+----+--------------+---------------+--------+
| 14 | Zara         | Ali           |  2000 |
| 15 | Daisy        | Das           |  5000 |
| 16 | John         | Paul          |  5000 |
| 17 | Mohd         | Yasee         |  3000 |
+----+--------------+---------------+--------+
4 rows in set (0.00 sec)
mysql>
```

Slika 5.1.2 Prikaz dobijene tabele EMPLOYEE

## 5.2 Zadaci za samostalni rad

### ZADATAK 5

*Cilj zadataka za samostalan rad je da student provežba tečeno znanje*

Po uzoru na primer objašnjen na https://netbeans.org/kb/docs/java/hibernate-java-se.html napraviti odgovarajuću JavaFX aplikaciju koristeći Criteria API

# Poglavlje 6

# Korišćenje SQL u Hibernate okruženju

## PRIMENA SQL UPISA U HIBERNATE OKRUŽENJU

*Vaša aplikacija će kreirati originalne SQL upite iz Session objekta primenom metoda createSQLQuery(). SQL rezultat možete spojiti sa HQL rezultatom primenom metoda addEntity().*

Iako radite u Hibernate okruženju, vi možete da koristite i originalne SQL upite da bi iskoristili neke specifična svojstva baze podataka, kao što je korišćenje CONNECT ključna reč kod Oracle RDBMS. Hiberbate 3.x dozvoljava primenu SQL upita, uključujući i primenu memorisanih procedura, kao i operacije create, update, delete i load za sve slogove.

Vaša aplikacija će kreirati originalne SQL upite iz Session objekta primenom metoda ***createSQLQuery().***

```
public SQLQuery createSQLQuery(String sqlString) throws HibernateException
```

Posle ubacivanja stringa sa SQL upitom (sa metodom createSQLQuery() možete povezati dobijeni SQL rezultat sa postojećim HQL rezultatom primenom metoda: ***addEntity(), addJoin(),*** i ***addScalar()***

**SQL skalarni upiti:**

Najednostavniji SQL upit obezbeđuje dobijanje liste skalarnih veličina iz jedne ili više tabela. Evo sintakse takvog SQL upita::

```
String sql = "SELECT first_name, salary FROM EMPLOYEE";
SQLQuery query = session.createSQLQuery(sql);
query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
List results = query.list();
```

**SQL upiti entiteta:**

SQL upit entiteta obezbeđuje dobijanje entiteskih objekata primenom metoda ***addEntity().***

```
String sql = "SELECT * FROM EMPLOYEE";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
List results = query.list();
```

**Imenovani SQL upiti:**

Sintaksa za dobijanje entitetskih objekata primenom originalnog SQL upita primenom ***addEntity()*** metoda i imenovanog SQL upita izgleda ovako:

```
String sql = "SELECT * FROM EMPLOYEE WHERE id = :employee_id";
SQLQuery query = session.createSQLQuery(sql);
query.addEntity(Employee.class);
query.setParameter("employee_id", 10);
List results = query.list()
```

## 6.1 Pokazni primeri

### PRIMER 9 - PRIMENA SQL-A U HIBERNATE-U

*Problem se definiše POJO klasom Employee, kreiranjem tabele u bazi EMPLOYEE i XML fajla za mapiranje objekata Employee u slogove (redove) tabele EMPLOYEE.*

Pođimo od sledeće POJO klase Employeee:

```
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
```

```
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
   public void setSalary( int salary ) {
      this.salary = salary;
   }
}
```

Kreirajmo tabelu EMPLOYEE

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Kreirajmo XML fajl za mapiranje:

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

<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
   </class>
</hibernate-mapping>
```

## PRIMER 9 - PRIMER PRIMENE SQL U HIBERNATE-U - KLASA APLIKACIJE

*Klasa apalikacije ManageEmployee sadrži metod main() sadrži originalne SQL upite*

Sada se kreira aplikaciona klasa **ManageEmployee** sa metodom ***main()*** koji sadrži originalne SQL upite.

```
import java.util.*;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.SQLQuery;
import org.hibernate.Criteria;
import org.hibernate.Hibernate;
import org.hibernate.cfg.Configuration;

public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      try{
         factory = new Configuration().configure().buildSessionFactory();
      }catch (Throwable ex) {
         System.err.println("Failed to create sessionFactory object." + ex);
         throw new ExceptionInInitializerError(ex);
      }
      ManageEmployee ME = new ManageEmployee();

      /* Add few employee records in database */
      Integer empID1 = ME.addEmployee("Zara", "Ali", 2000);
      Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
      Integer empID3 = ME.addEmployee("John", "Paul", 5000);
      Integer empID4 = ME.addEmployee("Mohd", "Yasee", 3000);

      /* List down employees and their salary using Scalar Query */
      ME.listEmployeesScalar();

      /* List down complete employees information using Entity Query */
      ME.listEmployeesEntity();
   }
   /* Method to CREATE an employee in the database */
   public Integer addEmployee(String fname, String lname, int salary){
      Session session = factory.openSession();
      Transaction tx = null;
      Integer employeeID = null;
      try{
         tx = session.beginTransaction();
         Employee employee = new Employee(fname, lname, salary);
         employeeID = (Integer) session.save(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
```

```
      return employeeID;
   }

   /* Method to  READ all the employees using Scalar Query */
   public void listEmployeesScalar( ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         String sql = "SELECT first_name, salary FROM EMPLOYEE";
         SQLQuery query = session.createSQLQuery(sql);
         query.setResultTransformer(Criteria.ALIAS_TO_ENTITY_MAP);
         List data = query.list();

         for(Object object : data)
         {
            Map row = (Map)object;
            System.out.print("First Name: " + row.get("first_name"));
            System.out.println(", Salary: " + row.get("salary"));
         }
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }

   /* Method to  READ all the employees using Entity Query */
   public void listEmployeesEntity( ){
      Session session = factory.openSession();
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         String sql = "SELECT * FROM EMPLOYEE";
         SQLQuery query = session.createSQLQuery(sql);
         query.addEntity(Employee.class);
         List employees = query.list();

         for (Iterator iterator =
                           employees.iterator(); iterator.hasNext();){
            Employee employee = (Employee) iterator.next();
            System.out.print("First Name: " + employee.getFirstName());
            System.out.print("  Last Name: " + employee.getLastName());
            System.out.println("  Salary: " + employee.getSalary());
         }
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
```

```
        }
    }
}
```

## PRIMER 9 - PRIMER PRIMENE SQL U HIBERNATE-U - KOMPILACIJA I IZVRŠENJE

*Kreiran fajl sa izvršnim kodom ManageEmployee.java se prevodi (kompilira) i dobijeni izvršni (binarni) program se izvršava. Dobijena je tabela EMPLOYEE sa trajnim objektima Employee.*

**Kompilacija i izvršenje:**
Prvo, treba da podesimo PATH i CLASSPATH parametre pre nego što pristupimo proceduri kompilacije i izvršenju programa. Zatim, primenjujemo preporučene korake kompilacije i izvršenja programa.

1. Kreiranje hibernate.cfg.xml fajla konfiguracije (urađeno ranije)
2. Kreiranje Employee.hbm.xml fajla za mapiranje (ko što je već prikazano)
3. Kreiranje ManageEmployee.java fajla sa izvornim kodom (kao što je već urađeno) i njena kompilacija
4. Izvršenje ManageEmployee izvršnog (binarnog) programa .

Trebalo bi da dobijete sledeći razultat.

```
$java ManageEmployee
 .....prikaz različitih LOG poruka.....
First Name: Zara, Salary: 2000 First Name: Daisy, Salary: 5000
First Name: John, Salary: 5000 First Name: Mohd, Salary: 3000
First Name: Zara Last Name: Ali Salary: 2000
First Name: Daisy Last Name: Das Salary: 5000
First Name: John Last Name: Paul Salary: 5000
First Name: Mohd Last Name: Yasee Salary: 3000
```

Slika 6.1.1 Prikaz dobijenog rezultata SQL upita

Ako proverite table EMPLOYEE, dobili bi sledeći prikaz dobijene tabele:

```
mysql> select * from EMPLOYEE;
+----+--------------+---------------+---------+
| id | first_name | last_name | salary |
+----+--------------+---------------+---------+
| 14 | Zara | Ali | 2000 |
| 15 | Daisy | Das | 5000 |
| 16 | John | Paul | 5000 |
| 17 | Mohd | Yasee | 3000 |
+----+--------------+---------------+---------+
4 rows in set (0.00 sec)
mysql>
```

Slika 6.1.2 Prikaz dobijene tabele EMPLOYEE

## 6.2 Zadaci za samostalni rad

### ZADATAK 6

*Cilj zadataka je da student primeni stečeno znanje*

Kreirati bazu podataka koja sadrži tabelu kupac, proizvod i narudžbina, a da odgovara opisu klasa da Kupac ima svoje ime, prezime. Proizvod ima svoj naziv i cenu. Narudžbina ima svog Kupca i listu narudžbina.

Nakon toga kreirati i konfigurisati Hiberante projekat koji će biti povezan sa kreiranom bazom podataka. Potrebno je da se izgenerišu Kupac, Proizvod i Narudžbina kao što je urađeno na vežbama.

Zatim kreirati klasu koja će realizovati CRUD operacije nad bazom korišćenjem SQL upita. Pored CRUD operacija dodati i metodu koja pronalazi kupca sa najvećom potrošnjom

Nakon toga je potrebno kreirati CRUD JavaFX formu za dodavanje novih proizvoda i kupaca, kao i prozor koji će omogućavati da se proizvod doda u narudžbinu.

# Poglavlje 7

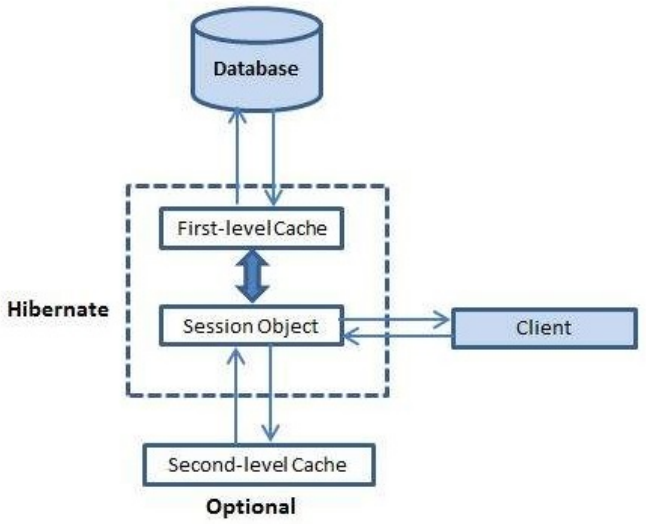
# Hibernate - keširanje

## DVA NIVOA KEŠIRANJA

*Keširanje (Caching) optimizuje performanse aplikacije. Prvi nivo keša je obavezan i radi sa Session objektom. Drugi nivo keša je opcioni i koristi se u radu sa više sesija.*

Keširanje (Caching) optimizira performanse aplikacije, i funkcionalno se smešta između vaše aplikacije i i baze podataka, a sa ciljem da se minimizira komunikacija sa bazom podataka (koja je na disku) i na taj način poboljšaju performanse (brzina rada) kod kritičkih aplikacija kod kojih su performanse bitne.

Keširanje kod Hibernate-a upotrebljava šemu keširanja u više nivoa, kao što je prikazano na slici 1 i objašnjeno u daljem tekstu.




Slika 7.1 Dva nivoa keširanja

Prvi nivo keša: Prvi nivo keša je keš Session objekta i ovo je obavezni keš kroz koji moraju da prođu svi zahtevi. Session objekat zadržava objekat pod svojom kontrolom pre nego što ga preda bazi podataka. Ako koristite promenu više performansi (atributa) jednog objekta, Hibernate pokušava da odloži promenu što duže da bi smanjio broj SQL iskaza. Kada zatvorite Session objekat, svi objekti koje je on keširao se gube.

Drugi nivo keša: Drugi nivo keša je opcioni keš. Pri pokušaju smeštaja nekog objekta u drugi nivo keša, vrši se prethodna provera prvog nivoa keša. Drugi nivo keša se konfiguriše prema

klasi ili prema kolekciji i najviše se korististi u slučaju rada sa više sesija, tj. od strane više Session objekata.

Svaki keš neke treće strane se može koristiti u Hibernate-u. Obezbeđen je za to interfajs ***org.hibernate.cache.CacheProvider***

Keš na nivou upita**:** U bliskoj integraciji sa drugim nivoom, obezbeđen je i opcioni keš objekata dobijenih HQL upiima.Koristi se najviše kod upita koji često ponavljaju iste parametre.

# DRUGI NIVO KEŠA

*Postoje četiri strategije konkurentnog rada sa kešom na drugom nivou: transactional, read-write, nonstrict-read-write i read-only.*

Hibernate uvek koristi keširanje prvog nivoa. Međutim, od vas zavisi primena keša drugog nivoa, jer je opcioni. Neki put i poboljšava performanse te ga onda treba isključiti. Treba da odlučite koju ćete strategiju konkuretnog rada da primenite i da konfigurišete prestanak keširanja i atribute fizičkog keširanja primenom keš provider-a.

**Startegije konkurentnog rada:**

Stategija konkurentnog rada je mediator koji je odgovoran za memorisanje podataka u kešu i njihovo ponovno dobijanje iz keša. Treba da odlučite koju strategiju ćete primeniti za svaku trajnu klasu ili kolekciju.

- ***Transactional*****:** Samo čita podatke - koristi se kada je važno da se spreči prebacivanje podataka o stanju pri konkurentnim transakcijama.
- ***Read-write*****:** Podržava i čitanje i pisanje. Upotreba kao i kod ***transactional***
- ***Nonstrict-read-write:*** Ne garantuje konsistentnost između keša i baze. Koristite kada se podaci retko menjaju..
- ***Read-only***: U slučajevima kada se podaci nikada ne menjaju.

Kao primer, upotrebićemo drugi nivo keša za slučaj naše klase **Employee.** Treba sada dodati fajl za mapiranje da bi saopštili Hibernate-u da kešira objekte **Employee** klase upotrebom ***read-write*** strategije.

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

<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <cache usage="read-write"/>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
```

```
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
   </class>
</hibernate-mapping>
```

# KEŠ PROVAJDER

*Keš provajder vrši keširanje aplikacije. Postoje četiri tipa keš provajdera. Pri izboru se mora voditi računa o njihovoj kompatibilnosti sa strategijama konkurentnosti.*

Posle određivanja strategije konkurentnosti, potrebno je da odredite provajdera za keširanje vaše aplikacije. Na slici su prikazana četiri moguća provajdera:

| RB | Naziv keša | Opis |
|---|---|---|
| 1 | EHCache | Pravi keš u memoriji ili disku i vrši klaster keširanje, a podržava i opcioni keš rezutata HQL upita. |
| 2 | OSCache | Podržava keširanje u memoriji i na disku u okviru jedne JVM, sa različitim opcijama trajanja i podrške kešu upita. |
| 3 | warmCache | Klaster keš baziran na Jgroups. Uotrebljava poništavanje klastera , ali ne podržava keširanje Hibetrnate upita. |
| 4 | JBoss Cache | Transakciona replika klestrizovanog keša baziranog na Jcroups multicast biblioteku. Podržava repiciranje ili poništavanje, sinhronu ili asinhron komiunikaciju, i optimističko i pesimističko zaključivanje tabela. Podržava i keš Hibernate upita. |

Slika 7.2 Četiri keš provajdera

Svaki keš provajder nije kompatibilan sa svakom strategijom konkurentnosti. Na slici 3. dat je pregled komptibilnih provajdera i strategija. Pri izboru provajdera, morate voditi računa o ovoj kompatibilnosti.

| Strategy/Provider | Read-only | Nonstrictread-write | Read-write |
|---|---|---|---|
| EHCache | X | X | X |
| OSCache | X | X | X |
| SwarmCache | X | X | |
| JBoss Cache | X | | |

Slika 7.3 Kompatibilnost keš provajdera i strategija konkurentnosti

Izabrani keš provajder treba da unesete u konfiguracioni fajl ***hibernate.cfg.xml*** . U sledećem primeru, mi smo izabrali EHCash za keš provajder drugog nivoa:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>

   <!-- Assume students is the database name -->
   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/test
   </property>
   <property name="hibernate.connection.username">
      root
   </property>
   <property name="hibernate.connection.password">
      root123
   </property>
   <property name="hibernate.cache.provider_class">
      org.hibernate.cache.EhCacheProvider
   </property>

   <!-- List of XML mapping files -->
   <mapping resource="Employee.hbm.xml"/>

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

# PRIMER 10 - KORIŠĆENJE PROVAJDERA EHCASHE

*Hibernate koristi EHCashe provajder uvek kada birate Emlpoyee objekte ili kada zovete Employee objekte iz baze koristeći indentifikator ovih objekata.*

Sada je potrebno da specificirate svojstva keša regiona. EHCache ima svoji posebni fajl konfigurisanja ***ehcashe.xml*** koji treba da navedete u ***CLASSPATH*** vaše aplikacije. Dole se daje izgled keš konfiguracije u ***ehcashe.xml*** za klasu **Employee**:

```
<diskStore path="java.io.tmpdir"/>
<defaultCache
```

```
maxElementsInMemory="1000"
eternal="false"
timeToIdleSeconds="120"
timeToLiveSeconds="120"
overflowToDisk="true"
/>

<cache name="Employee"
maxElementsInMemory="500"
eternal="true"
timeToIdleSeconds="0"
timeToLiveSeconds="0"
overflowToDisk="false"
/>
```

Sada imamo osposobljen drugi nivo keša za klasu **Employee.** Hibernate koristi ovaj keš uvek kada birate **EmIpoyee** objekte ili kada zovete **Employee** objekte iz baze koristeći indentifikator ovih objekata.

Preporučuje se da uporedite performanse aplikacije pre i posle keširanja. Koristite keširanje samo ako su sa njim performanse bolje.

**Keš na nivou upita:**

Da bi koristili keš upita, potrebno je da ga prvo aktivirate upotrebom ***.cache.use_query_cache="true"*** svojstva u konfikuracionom fajlu. Na ovaj način Hibernate kreira potrebne kešove u memoriji i zadržava upite i njihove indetifikatore. Koristite ***query cache, metod setCacheable(Boolean***) u Queryklasi.

```
Session session = SessionFactory.openSession();
Query query = session.createQuery("FROM EMPLOYEE");
query.setCacheable(true);
List users = query.list();
SessionFactory.closeSession();
```

HIbernate podržava keš regiona. Oni su deo keša sa posebnim nazivom.

```
Session session = SessionFactory.openSession();
Query query = session.createQuery("FROM EMPLOYEE");
query.setCacheable(true);
query.setCacheRegion("employee");
List users = query.list();
SessionFactory.closeSession()
```

Hibernate skladišti i traži upite u regionu keša gde su podaci o zaposlenim.

# Poglavlje 8

# Hibernate - paketna obrada

## ZAŠTO JE POTREBNA PAKETNA OBRADA?

*Kod keširanja na prvom nivou zbog velikog broja slogova, ona može biti nedovoljna da ih preuzme zbog javljanja izuzetka OutOfMemoryException: Ovaj problem rešava paketna obrada.*

Razmotrimo situaciju kada treba da stavljate veliki broj slogova u bazu upotrebom Hibernate-a. Taj posao obavlja sledeći program:

```
Session session = SessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Employee employee = new Employee(.....);
    session.save(employee);
}
tx.commit();
session.close();
```

Po pravilu, Hibernate će kaširati sve trajne objekte u prvom nivou keširana (na nivou sesije) i ***vaša memorija će biti nedovoljna da ih preuzme zbog javljanja izuzetka OutOfMemoryException***: Ovaj problem možete rešiti primenom paketne obrade (batch processing) sa Hibernate-om.

Da bi koristili paketnu obradu. prvo podesite Hibernate sa ***jdbc.batch_size*** gde je veličina paketa 20 ili 50, zavisno od veličine objekta. Na osnovu ovoga, Hibernate kontejner će na svakhh X redova (slogova) ubaciti po jedan paket za obradu. Ovo traži malu promenu u Vašem kodu:

```
Session session = SessionFactory.openSession();
Transaction tx = session.beginTransaction();
for ( int i=0; i<100000; i++ ) {
    Employee employee = new Employee(.....);
    session.save(employee);
    if( i % 50 == 0 ) { // Same as the JDBC batch size
        //flush a batch of inserts and release memory:
        session.flush();
        session.clear();
    }
```

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

Ukoliko koristite UPDATE opciju, morate koristi sledeći kod:

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

ScrollableResults employeeCursor = session.createQuery("FROM EMPLOYEE")
                                   .scroll();
int count = 0;

while ( employeeCursor.next() ) {
   Employee employee = (Employee) employeeCursor.get(0);
   employee.updateEmployee();
   seession.update(employee);
   if ( ++count % 50 == 0 ) {
      session.flush();
      session.clear();
   }
}
tx.commit();
session.close();
```

# 8.1 Pokazni primeri

## PRIMER 11 - PAKETNE OBARDE

*Paketna obrada se specificira sa svojstvom hibernate.jdbc.batch_size*

Dodajmo u ranije kreirani konfiguracioni fajl ***hibernate.jdbc.batch_size property***

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration SYSTEM
"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
   <session-factory>
   <property name="hibernate.dialect">
      org.hibernate.dialect.MySQLDialect
   </property>
   <property name="hibernate.connection.driver_class">
      com.mysql.jdbc.Driver
   </property>
```

```
   <!-- Assume students is the database name -->
   <property name="hibernate.connection.url">
      jdbc:mysql://localhost/test
   </property>
   <property name="hibernate.connection.username">
      root
   </property>
   <property name="hibernate.connection.password">
      root123
   </property>
   <property name="hibernate.jdbc.batch_size">
      50
   </property>

   <!-- List of XML mapping files -->
   <mapping resource="Employee.hbm.xml"/>

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

## PRIMER 11 - PRIMER PAKETNE OBRADA - POJO KLASA EMPLOYEE

### *POJO klasa se mora definisati*

Definišimo sledeću POJO Emlpoyee klasu:

```
public class Employee {
   private int id;
   private String firstName;
   private String lastName;
   private int salary;

   public Employee() {}
   public Employee(String fname, String lname, int salary) {
      this.firstName = fname;
      this.lastName = lname;
      this.salary = salary;
   }
   public int getId() {
      return id;
   }
   public void setId( int id ) {
      this.id = id;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName( String first_name ) {
      this.firstName = first_name;
   }
```

```
   public String getLastName() {
      return lastName;
   }
   public void setLastName( String last_name ) {
      this.lastName = last_name;
   }
   public int getSalary() {
      return salary;
   }
   public void setSalary( int salary ) {
      this.salary = salary;
   }
}
```

## PRIMER 11 - PRIMER PAKETNE OBRADA - KLASA APLIKACIJE BEZ KEŠIRANJA

*Primenom metoda flush() i clear() Session objekta, Hibernate zapisuje slogove direktno u bazu, bez prethodnog keširanja.*

Kreirajmo **EMPLOYEE** tabelu za smeštaj Employee objekata:

```
create table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

Kreirajmo fajl mapiranja **Employee** objekata u **EMPLOYEE** tabelu:

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

<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
```

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

A sada, kreirajmo klasu aplikacije sa ***main()*** metodom koji koristi ***flush()*** i ***clear()*** metode objekta Session, tako da Hibernate zapisuje ove slogove u bazu podataka umesto da ih kešira u memoriji.

```
import java.util.*;

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

public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      try{
         factory = new Configuration().configure().buildSessionFactory();
      }catch (Throwable ex) {
         System.err.println("Failed to create sessionFactory object." + ex);
         throw new ExceptionInInitializerError(ex);
      }
      ManageEmployee ME = new ManageEmployee();

      /* Add employee records in batches */
      ME.addEmployees( );
   }
   /* Method to create employee records in batches */
   public void addEmployees( ){
      Session session = factory.openSession();
      Transaction tx = null;
      Integer employeeID = null;
      try{
         tx = session.beginTransaction();
         for ( int i=0; i<100000; i++ ) {
            String fname = "First Name " + i;
            String lname = "Last Name " + i;
            Integer salary = i;
            Employee employee = new Employee(fname, lname, salary);
            session.save(employee);
            if( i % 50 == 0 ) {
               session.flush();
               session.clear();
            }
         }
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
```

```
      }finally {
         session.close();
      }
      return ;
   }
}
```

## PRIMER 11 - PRIMER PAKETNE OBRADA - KOMPILACIJA I IZVRŠENJE

*Data je procedura pripreme programa za kompilaciju i izvršenje.*

**Kompilacija i izvršenje:**

Ovde se daju koraci do kompilacije i izvršenja programa u slučaju prethodne aplikacije. Pretpostavljamo da ste podeili prethodno PATH i CLASSPATH na pravi način pre ove procedure kompilirnjai izvršenja:

1. Kreirajte ***cfg.xml*** konfiguracioni fajl, kao što je već pokazano.
2. Krirajte ***Employee.hbm.xml*** fajl mapiranja kao što je već pokazano. .
3. Kreirajte ***Employee.java*** fajl sa izvornim kodom i onda ga kampirajte.
4. Kreirajte ***ManageEmployee.java*** izvorni fajl jao što je pokazano i kompilirajte ga. kao što ej već pokazano.
5. IZVRŠITE **ManageEmployee** binarni program ta kreira 100000 slogova u tabeli **EMPLOYEE.**

# 8.2 Zadaci za samostalni rad

## ZADACI ZA SAMOSTALNI RAD STUDENTA

*Cilj je provežbavanje stečenog znanja*

**Zadatak 7:**

Doraditi Zadatak 4 tako da koristi paketnu obradu po uzoru na primer 11.

# Poglavlje 9

# Hibernate - Presretači (Inteceptors)

## INTERCEPTOR INTERFACE

*Interceptor interfejs obezbeđuje metode za različita stanja objekta da bi obavili zahtevane zadatke.*

Pri kreiranju objekta ili pri njegovoj promeni, mora da se uskladište u trajnoj memoriji (npr. disku), tj. u bazi podataka. Kada je objekat potreban, on se poziva iz baze i stavlja u memoriju aplikacije.

Objekat na ovaj način prolazi kroz različita stanja u svom životnom ciklusu. ***Interceptor*** interfejs obezbeđuje metode koji se pozivaju u različitim stanjima objekta da bi obavili zahtevane zadatke. Ovi metodi se pozivaju iz sesije u aplikaciju, dozvoljavajući aplikaciji da proverava sadržaj i manipuliše svojstvima trajnog objekta pre nego što se uskladišti ponovo, ili dok se ne pomeni, izbriše ili dovede u memorju. ***Interceptor*** interfejs obezbeđuje sledeće metode:

1 ***findDirty()*** - poziva se kada i metod ***flush()*** u Session objektu.

2 ***instantiate()*** - poziva se radi dobijanja objekta trajne klase
3 ***isUnsaved()*** - poziva prikom korišćenja metoda ***saveOrUpdate()***
4 ***onDelete()*** - poziva se pre brisanja objekta.
5. ***onFlushDirty()*** - poyiva se kad ahiberbet nađe prljav objekat za vreme operacije menjanja objekta.
6 ***onLoad()*** - poziva se pre inicijalizacije objekta

7. ***onSave()*** - poziva se pre uksladišćenja objekta

8. ***postFlush()*** - poziva se posle promene u objektu u memoriji

Hibernate presretač (***Inreceptor*** API) daje nam potpunu kontrolu nad izgledom objekta u aplikaciji i u bazi podataka.

**Kako koristiti presretače?**

Da bi napravili presretač, treba da direktno primenite klasu ***Interceptor*** ili da proširite klasu ***EmptyInterceptor***. U daljem tekstu se navode koraci upotrebe funkcija Hibernate Interceptor Interfejsa:

**Kreiranje prestretača:**

Proširićemo funkcionalnost ***EmptyInterceptor*** u našem primeru. Metod ***Interceptor*** interfejsa će se automatski pozivati kada se objekat **Employee** kreira i kada se menja.. Možete dodati više metoda u skladu sa novim zahtevima.

```
import java.io.Serializable;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.EmptyInterceptor;
import org.hibernate.Transaction;
import org.hibernate.type.Type;

public class MyInterceptor extends EmptyInterceptor {
   private int updates;
   private int creates;
   private int loads;

   public void onDelete(Object entity,
                    Serializable id,
                    Object[] state,
                    String[] propertyNames,
                    Type[] types) {
       // do nothing
   }

   // This method is called when Employee object gets updated.
   public boolean onFlushDirty(Object entity,
                    Serializable id,
                    Object[] currentState,
                    Object[] previousState,
                    String[] propertyNames,
                    Type[] types) {
       if ( entity instanceof Employee ) {
          System.out.println("Update Operation");
          return true;
       }
       return false;
   }
   public boolean onLoad(Object entity,
                   Serializable id,
                   Object[] state,
                   String[] propertyNames,
                   Type[] types) {
       // do nothing
       return true;
   }
   // This method is called when Employee object gets created.
   public boolean onSave(Object entity,
                   Serializable id,
                   Object[] state,
                   String[] propertyNames,
                   Type[] types) {
       if ( entity instanceof Employee ) {
          System.out.println("Create Operation");
          return true;
       }
       return false;
```

```
   }
   //called before commit into database
   public void preFlush(Iterator iterator) {
      System.out.println("preFlush");
   }
   //called after committed into database
   public void postFlush(Iterator iterator) {
      System.out.println("postFlush");
   }
}
```

# 9.1 Pokazni primeri

## PRIMER 12 - KREIRANJE POJO KLASE, TABELE EMPLOYEE I FAJLA MAPIRANJA

*Za datu POJO klasu, definiše se tabela baze a fajlom mapiranja vrši se povezivanje, tj. mapiranje klase Emlopyee u tabelu EMPLOYEE.*

**Kreiranje POJO klasa :**

Sada ćemo modifikovati raniji primer u kome smo koristili EMPLOYEE tabelu i Employee klasu.

```
reate table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

**Kreiranje tabela baze podataka::**

Kao što je poznato, drugi korak je kreiranje tabela vaše baze podataka. Imamo jednu tabelu za svaku klasu koju želite da bude trajna. Pretpostavimo da objekti treba da budu smešteni, kao slogovi (redovi) u sledećoj tabeli:

```
reate table EMPLOYEE (
   id INT NOT NULL auto_increment,
   first_name VARCHAR(20) default NULL,
   last_name  VARCHAR(20) default NULL,
   salary     INT  default NULL,
   PRIMARY KEY (id)
);
```

**Kreiranje konfiguracionog fajla mapiranja:**

Fajl mapiranja daje instrukcije Hibernate-u kako da mapira definisanu klasu u tabele baze podataka.

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

<hibernate-mapping>
   <class name="Employee" table="EMPLOYEE">
      <meta attribute="class-description">
         This class contains the employee detail.
      </meta>
      <id name="id" type="int" column="id">
         <generator class="native"/>
      </id>
      <property name="firstName" column="first_name" type="string"/>
      <property name="lastName" column="last_name" type="string"/>
      <property name="salary" column="salary" type="int"/>
   </class>
</hibernate-mapping
```

## PRIMER 12 - KREIRANJE APLIKACIONE KLASE

### *Metod main(), prilikom korišćenja Session objekta, koristi Interceptor klasu kao argument.*

**Kreiranje aplikacione klase:**

Sada kreirajmo aplikacionu klasu sa ***main()*** metodom koji izvršava aplikaciju. Zapazite da sada, prilikom korišćenja Session objekta koristimo našu Interceptor klasu kao argument.

```
import java.util.List;
import java.util.Date;
import java.util.Iterator;

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

public class ManageEmployee {
   private static SessionFactory factory;
   public static void main(String[] args) {
      try{
         factory = new Configuration().configure().buildSessionFactory();
      }catch (Throwable ex) {
         System.err.println("Failed to create sessionFactory object." + ex);
         throw new ExceptionInInitializerError(ex);
```

```
    }

    ManageEmployee ME = new ManageEmployee();

    /* Add few employee records in database */
    Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);
    Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
    Integer empID3 = ME.addEmployee("John", "Paul", 10000);

    /* List down all the employees */
    ME.listEmployees();

    /* Update employee's records */
    ME.updateEmployee(empID1, 5000);

    /* Delete an employee from the database */
    ME.deleteEmployee(empID2);

    /* List down new list of the employees */
    ME.listEmployees();
}
/* Method to CREATE an employee in the database */
public Integer addEmployee(String fname, String lname, int salary){
    Session session = factory.openSession( new MyInterceptor() );
    Transaction tx = null;
    Integer employeeID = null;
    try{
        tx = session.beginTransaction();
        Employee employee = new Employee(fname, lname, salary);
        employeeID = (Integer) session.save(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
    return employeeID;
}
/* Method to  READ all the employees */
public void listEmployees( ){
    Session session = factory.openSession( new MyInterceptor() );
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        List employees = session.createQuery("FROM Employee").list();
        for (Iterator iterator =
                          employees.iterator(); iterator.hasNext();){
           Employee employee = (Employee) iterator.next();
           System.out.print("First Name: " + employee.getFirstName());
           System.out.print("  Last Name: " + employee.getLastName());
           System.out.println("  Salary: " + employee.getSalary());
        }
```

```
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to UPDATE salary for an employee */
   public void updateEmployee(Integer EmployeeID, int salary ){
      Session session = factory.openSession( new MyInterceptor() );
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                    (Employee)session.get(Employee.class, EmployeeID);
         employee.setSalary( salary );
         session.update(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
   /* Method to DELETE an employee from the records */
   public void deleteEmployee(Integer EmployeeID){
      Session session = factory.openSession( new MyInterceptor() );
      Transaction tx = null;
      try{
         tx = session.beginTransaction();
         Employee employee =
                   (Employee)session.get(Employee.class, EmployeeID);
         session.delete(employee);
         tx.commit();
      }catch (HibernateException e) {
         if (tx!=null) tx.rollback();
         e.printStackTrace();
      }finally {
         session.close();
      }
   }
}
```

## PRIMER 12 - KOMPILACIJA I IZVRŠENJE

*Vrši se kompilacija izvornih fajlova Emmployee.java, MyIntraceptor.java i ManageEmployee.java, a izvršenje binarnog fajla ManageEmployee.*

**Kompilacia i izvršenje:**

Ovde se daju koraci do kompilacije i izvršenja programa u slučaju prethodne aplikacije. Pretpostavljamo da ste podeili prethodno PATH i CLASSPATH na pravi način pre ove procedure kompiliranja i izvršenja:

1. Kreirajte ***cfg.xml*** konfiguracioni fajl, kao što je već pokazano.
2. Kreirajte ***Employee.hbm.xml*** fajl mapiranja kao što je već pokazano.
3. ***Kreirajte Employee.java*** fajl sa izvornim kodom i onda ga kampirajte.
4. Kreirajte ***MyInreceptor.java*** izvorni fajl koji je već prikazan
5. Kreirajte ***ManageEmployee.java*** izvorni fajl jao što je pokazano i kompilirajte ga. kao što ej već pokazano.
6. Izvršite ***ManageEmployee***binarni fajl. .

Na slici je prikazan rezultat koji se dobija izvršenjem ove aplikacije.

```
$java ManageEmployee
 .....OVDE SE PRIKAZUJU LOG PORUKE.....
Create Operation
preFlush
postFlush
Create Operation
preFlush
postFlush
Create Operation
preFlush
postFlush
First Name: Zara Last Name: Ali Salary: 1000
First Name: Daisy Last Name: Das Salary: 5000
First Name: John Last Name: Paul Salary: 10000
preFlush
postFlush
preFlush
Update Operation
postFlush
preFlus
postFlush
First Name: Zara Last Name: Ali Salary: 5000
First Name: John Last Name: Paul Salary: 10000
preFlush
postFlush
```

Slika 9.1.1 Prikaz rezultata izvršenja

# PRIMER 12 - DOBIJENA TABELA EMPLOYEE

## *Prikaz dobijene tabele EMPLOYEE*

Ako proverite vašu EMPLOYEE tabelu, dobićete sledeći prikaz::

```
mysql> select * from EMPLOYEE;
+----+--------------+---------------+---------+
| id  | first_name | last_name | salary |
+----+---------------+--------------+--- -----+
| 29 | Zara          | Ali           |  2000 |
| 31 | John          | Paul          |  5000 |
+----+--------------+--------------+---------+
4 rows in set (0.00 sec)
mysql>
```

Slika 9.1.2 Tabela EMPLOYEE

# Poglavlje 10

# Studija slučaja - StudentService

## KREIRANJE HIBERNATE PROJEKTA

*Cilje sekcije da se prikaže kako se dodaju Hibernate biblioteke*

Da bi smo kreirali Hibernate projekat potrebno je prvo da kreiramo običan Java projekat ili u našem slučaju JavaFX projekat koji ćemo nazvati KI203-V11. Nakon toga pritisnemo desni klik na projekat, u padajućem meniju odaberemo opciju Properties nakon čega će nam se pojaviti prozor kao na slici 1.

Slika 10.1 Dodavanje biblioteka u okviru NetBeans projekta

Sledeća stvar koju treba uraditi je da selektujemo Libraries opciju u meniju i da kliknemo na dugme Add Library sa desne strane. Nakon toga će nam se otvoriti prozor sa svim dostupnim datotekama odakle treba da odaberemo Hibernate 4.3.x i kliknemo na dugme Add Library (slika 2).

Slika 10.2 Dodavanje Hibernate biblioteke u NetBeans-u

## SQL SKRIPT ZA BAZU

*U nastavku je dat sadržaj cs102_baza.sql file-a*

Za Hibernate projekte je neophodno da imamo bazu podataka koja će u našem primeru imati tabele profesor i predmet. Profesor ima atribute id, ime, prezime i zvanje, dok predmet ima atribute id, sifra, naziv i espb. Tabele profesor i predmet su u relaciji jedan na više (jedan profesor više predmeta).

Ispod je data skripta za kreiranje potrebne baze podataka:

```
-- phpMyAdmin SQL Dump
-- version 4.5.1
-- http://www.phpmyadmin.net
--
-- Host: 127.0.0.1
-- Generation Time: May 19, 2016 at 08:52 PM
-- Server version: 10.1.9-MariaDB
-- PHP Version: 5.6.15

SET SQL_MODE = "NO_AUTO_VALUE_ON_ZERO";
```

```
SET time_zone = "+00:00";


/*!40101 SET @OLD_CHARACTER_SET_CLIENT=@@CHARACTER_SET_CLIENT */;
/*!40101 SET @OLD_CHARACTER_SET_RESULTS=@@CHARACTER_SET_RESULTS */;
/*!40101 SET @OLD_COLLATION_CONNECTION=@@COLLATION_CONNECTION */;
/*!40101 SET NAMES utf8mb4 */;

--
-- Database: `cs102_baza`
--

-- --------------------------------------------------------

--
-- Table structure for table `predmet`
--

CREATE TABLE `predmet` (
  `id` int(10) UNSIGNED NOT NULL,
  `sifra` varchar(10) NOT NULL,
  `naziv` varchar(100) NOT NULL,
  `espb` int(11) NOT NULL,
  `profesor_id` int(10) UNSIGNED NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

--
-- Dumping data for table `predmet`
--

INSERT INTO `predmet` (`id`, `sifra`, `naziv`, `espb`, `profesor_id`) VALUES
(1, 'CS322', 'Programiranje u C#', 5, 4),
(2, 'CS324', 'Skripting jezici', 6, 2),
(3, 'SE325', 'Upravljanje projektima', 6, 6),
(5, 'MA101', 'Matematika 1', 6, 7);

-- --------------------------------------------------------

--
-- Table structure for table `profesor`
--

CREATE TABLE `profesor` (
  `id` int(10) UNSIGNED NOT NULL,
  `ime` varchar(50) NOT NULL,
  `prezime` varchar(50) NOT NULL,
  `zvanje` varchar(50) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

--
-- Dumping data for table `profesor`
--
```

```
INSERT INTO `profesor` (`id`, `ime`, `prezime`, `zvanje`) VALUES
(1, 'Stevan', 'Jokic', 'docent'),
(2, 'Milan', 'Markovic', 'docent'),
(4, 'Luka', 'Pavic', 'redovni profesor'),
(6, 'Ana', 'Simic', 'redovni profesor'),
(7, 'Marija', 'Andric', 'docent');

--
-- Indexes for dumped tables
--

--
-- Indexes for table `predmet`
--
ALTER TABLE `predmet`
  ADD PRIMARY KEY (`id`),
  ADD KEY `profesor_id` (`profesor_id`),
  ADD KEY `profesor_id_2` (`profesor_id`),
  ADD KEY `profesor_id_3` (`profesor_id`);

--
-- Indexes for table `profesor`
--
ALTER TABLE `profesor`
  ADD PRIMARY KEY (`id`);

--
-- AUTO_INCREMENT for dumped tables
--

--
-- AUTO_INCREMENT for table `predmet`
--
ALTER TABLE `predmet`
  MODIFY `id` int(10) UNSIGNED NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=6;
--
-- AUTO_INCREMENT for table `profesor`
--
ALTER TABLE `profesor`
  MODIFY `id` int(10) UNSIGNED NOT NULL AUTO_INCREMENT, AUTO_INCREMENT=8;
--
-- Constraints for dumped tables
--

--
-- Constraints for table `predmet`
--
ALTER TABLE `predmet`
  ADD CONSTRAINT `predmet_ibfk_1` FOREIGN KEY (`profesor_id`) REFERENCES `profesor`
(`id`);

/*!40101 SET CHARACTER_SET_CLIENT=@OLD_CHARACTER_SET_CLIENT */;
/*!40101 SET CHARACTER_SET_RESULTS=@OLD_CHARACTER_SET_RESULTS */;
```

```
/*!40101 SET COLLATION_CONNECTION=@OLD_COLLATION_CONNECTION */;
```

Za korišćenje baze podataka nam je neophodan JDBC driver koji je korišćen i u prethodnim lekcijama sa bazama podataka.

# GENERISANJE ENTITET KLASA (PRVI DEO)

## *Cilj sekcije je da se prvo demonstrira povezivanje sa bazom podataka kroz NetBeans*

Prvo je potrebno da u okviru Services taba u NetBeans-u pod stavkom Databases lociramo našu bazu podataka, zatim kliknemo na nju desnim klikom i odaberemo opciju Connect u padajućem meniju. Nakon toga će nam se prikazati prozor koji nam traži da unesemo korisničko ime i šifru za pristup bazi (po default-u je username: root, password: "" - prazan string).

Slika 10.3 Lociranje baze u okvriu NetBeans servisa

Slika 10.4 Unos korisničkog imena i šifre za pristup bazi

Kada smo izvršili konektovanje na bazu možemo da izgenerišemo Java klase tj. entitete na osnovu strukture baze podataka.

# GENERISANJE ENTITET KLASA (DRUGI DEO)

## *Cilj sekcije je da se demonstrira generisanje entitet klasa.*

Na osnovu strukture tabela baze podataka moguće je da se generišu Java klase koje će zadržati relacije definisane u bazi podataka. Ove klase se nazivaju entitet klase i trebaju biti smeštene u paket model. Dakle, prvo kreiramo paket, zatim desnim klikom odaberemo stavku New u padajućem meniju, a zatim Entity Classes From Database nakon čega će nam se prikazati sledeći prozor.

Slika 10.5 Generisanje entitet klasa (prvi korak)

Nakon toga u sa leve strane možemo videti tabele iz baze podataka koje trebamo selektovati i dodati ih klikom na dugme add (slika 6). Nakon toga kliknemo next.

Slika 10.6 Generisanje entitet klasa (drugi korak)

# GENERISANJE ENTITET KLASA (TREĆI DEO)

*Cilj sekcije je da se demonstrira specifikacija generisanja entitet klasa.*

Kada smo dodali tabele iz baze podataka potrebno je da definišemo na koji način želimo da nam se generišu entitet klase. Odabrati opcije kao na slicli 7 (mogu se odabrati i sve opcije kao i ni jedna od ponuđenih).

Slika 10.7 Specifikacija generisanja entitet klasa

Nakon čekiranja potrebno je da kliknemo Next.

Nakon toga sledi druga faza specifikacije generisanja entitet klasa pri čemu je potrebno da odaberemo tip kolekcije koji će biti List kao i čekiraćemo opciju Use Column Names in Realationships (slika 8)

Slika 10.8 Finalni korak generisanja entiteta

Nakon odabira specifikacije potrebno je kliknuti na dugme Finish nakon čega će se dodati entitet klase Profesor i Predmet.

# HIBERNATE MAPPING FILES AND POJOS FROM DATABASE - PROFESOR

*U paketu model kreiramo klasu Profesor koja odgovara tabeli*

U entitet klasi Profesor možemo zapaziti da imamo sve atribute definisane u tabeli Profesor iz baze podataka kao i elemente POJO klase (konstruktore, gettere, settere, itd.) i Hibernate anotacije koje su pominjane na predavanjima. Takođe možemo videti da klasa Profesor sadrži kao atribut listu predmeta sa anotacijom @OneToMany na osnovu čega možemo zaključiti da su klase Profesor i Predmet u odgovarajućoj relaciji.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package model;

import java.io.Serializable;
import java.util.ArrayList;
import java.util.List;
import javax.persistence.Basic;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
```

```
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.OneToMany;
import javax.persistence.Table;

/**
 *
 * @author rados
 */
@Entity
@Table(name = "profesor")
@NamedQueries({
    @NamedQuery(name = "Profesor.findAll", query = "SELECT p FROM Profesor p"),
    @NamedQuery(name = "Profesor.findById", query = "SELECT p FROM Profesor p WHERE
p.id = :id"),
    @NamedQuery(name = "Profesor.findByIme", query = "SELECT p FROM Profesor p
WHERE p.ime = :ime"),
    @NamedQuery(name = "Profesor.findByPrezime", query = "SELECT p FROM Profesor p
WHERE p.prezime = :prezime"),
    @NamedQuery(name = "Profesor.findByZvanje", query = "SELECT p FROM Profesor p
WHERE p.zvanje = :zvanje")})
public class Profesor implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "id")
    private Integer id;
    @Basic(optional = false)
    @Column(name = "ime")
    private String ime;
    @Basic(optional = false)
    @Column(name = "prezime")
    private String prezime;
    @Basic(optional = false)
    @Column(name = "zvanje")
    private String zvanje;
    @OneToMany(cascade = CascadeType.ALL, mappedBy = "profesorId")
    private List predmetList;

    public Profesor() {
    }

    public Profesor(String ime, String prezime, String zvanje) {
        this.ime = ime;
        this.prezime = prezime;
        this.zvanje = zvanje;
    }

    public Profesor(Integer id, String ime, String prezime, String zvanje) {
```

```
        this.id = id;
        this.ime = ime;
        this.prezime = prezime;
        this.zvanje = zvanje;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getIme() {
        return ime;
    }

    public void setIme(String ime) {
        this.ime = ime;
    }

    public String getPrezime() {
        return prezime;
    }

    public void setPrezime(String prezime) {
        this.prezime = prezime;
    }

    public String getZvanje() {
        return zvanje;
    }

    public void setZvanje(String zvanje) {
        this.zvanje = zvanje;
    }

    public List getPredmetList() {
        return predmetList;
    }

    public void setPredmetList(List predmetList) {
        this.predmetList = predmetList;
    }

//    public void addPredmet(Predmet predmet){
//        if(predmetList == null){
//            predmetList = new ArrayList<>();
//        }
//        this.predmetList.add(predmet);
//    }
```

```
    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not
set
        if (!(object instanceof Profesor)) {
            return false;
        }
        Profesor other = (Profesor) object;
        if ((this.id == null
&amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;&amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;
other.id != null) || (this.id != null
&amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;&amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;
!this.id.equals(other.id))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return id + " " + ime + " " + prezime + ", " + zvanje;
    }
}
```

# HIBERNATE MAPPING FILES AND POJOS FROM DATABASE - PREDMET

## *U paketu model kreiramo klasu Predmet koja odgovara tabeli u bazi*

U entitet klasi Predmet možemo uvideti sve atribute iz tabele Predmet u bazi, odgovarajuće elemente POJO klase i Hibernate anotacije. U klasi predmet imamo atribut profesorId koji se odnosi na objekat klase Profesor. Dakle, ovde se još jednom potvrđuje da su prilikom generisanja entitet klasa zadržane relacije iz baze podataka.

Klasa Predmet:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package model;
```

```
import java.io.Serializable;
import javax.persistence.Basic;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.JoinColumn;
import javax.persistence.ManyToOne;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.Table;

/**
 *
 * @author rados
 */
@Entity
@Table(name = "predmet")
@NamedQueries({
    @NamedQuery(name = "Predmet.findAll", query = "SELECT p FROM Predmet p"),
    @NamedQuery(name = "Predmet.findById", query = "SELECT p FROM Predmet p WHERE
p.id = :id"),
    @NamedQuery(name = "Predmet.findBySifra", query = "SELECT p FROM Predmet p
WHERE p.sifra = :sifra"),
    @NamedQuery(name = "Predmet.findByNaziv", query = "SELECT p FROM Predmet p
WHERE p.naziv = :naziv"),
    @NamedQuery(name = "Predmet.findByEspb", query = "SELECT p FROM Predmet p WHERE
p.espb = :espb")})
public class Predmet implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Basic(optional = false)
    @Column(name = "id")
    private Integer id;
    @Basic(optional = false)
    @Column(name = "sifra")
    private String sifra;
    @Basic(optional = false)
    @Column(name = "naziv")
    private String naziv;
    @Basic(optional = false)
    @Column(name = "espb")
    private int espb;
    @JoinColumn(name = "profesor_id", referencedColumnName = "id")
    @ManyToOne(optional = false)
    private Profesor profesorId;

    public Predmet() {
    }
```

```
public Predmet(Integer id) {
    this.id = id;
}

public Predmet(String sifra, String naziv, int espb) {
    this.sifra = sifra;
    this.naziv = naziv;
    this.espb = espb;
}

public Predmet(String sifra, String naziv, int espb, Profesor profesorId) {
    this.sifra = sifra;
    this.naziv = naziv;
    this.espb = espb;
    this.profesorId = profesorId;
}

public Integer getId() {
    return id;
}

public void setId(Integer id) {
    this.id = id;
}

public String getSifra() {
    return sifra;
}

public void setSifra(String sifra) {
    this.sifra = sifra;
}

public String getNaziv() {
    return naziv;
}

public void setNaziv(String naziv) {
    this.naziv = naziv;
}

public int getEspb() {
    return espb;
}

public void setEspb(int espb) {
    this.espb = espb;
}

public Profesor getProfesorId() {
    return profesorId;
}
```

```
    public void setProfesorId(Profesor profesorId) {
        this.profesorId = profesorId;
    }

    @Override
    public int hashCode() {
        int hash = 0;
        hash += (id != null ? id.hashCode() : 0);
        return hash;
    }

    @Override
    public boolean equals(Object object) {
        // TODO: Warning - this method won't work in the case the id fields are not
set
        if (!(object instanceof Predmet)) {
            return false;
        }
        Predmet other = (Predmet) object;
        if ((this.id == null
&amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;&amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;am
p;amp; other.id != null) || (this.id != null
&amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;&amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;amp;am
p;amp; !this.id.equals(other.id))) {
            return false;
        }
        return true;
    }

    @Override
    public String toString() {
        return id + " " + sifra + " - " + naziv + ", \nprof: " +
profesorId.getIme() + " " + profesorId.getPrezime();
    }

}
```

# KREIRANJE HIBERNATE.CFG.XML

*File hibernate.cfg.xml se postavlja u root direktorijum aplikacije*

Nakon generisanja entiteta potrebno je da kreiramo hibernate.cfg.xml u koji predstavlja konfiguracioni fajl koji nam omogućava da se aplikacija poveže sa bazom podataka. Pored parametara za pristup bazi podataka potrebno je da u ovoj datoteci navedemo entitet klase na koje ćemo mapirati podatke iz tabela baze podataka.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC "-//Hibernate/Hibernate Configuration DTD
3.0//EN" "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
```

```
<hibernate-configuration>
     <session-factory>
        <property
name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
        <property
name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/
cs102_baza</property>
        <property name="hibernate.connection.username">root</property>
        <property name="hibernate.connection.password"></property>
        <property name="hibernate.show_sql">true</property>
        <!-- Mapiranje klasa -->
        <mapping class="model.Profesor"/>
        <mapping class="model.Predmet"/>
    </session-factory>
</hibernate-configuration>
```

# HIBERNETUTIL.JAVA

*Hibernate Utility class with a convenient method to get Session Factory object.*

Sledeća klasa koju kreiramo će biti HibernateUtil koja nam obezbeđuje sesiju prema bazi podataka i time omogućava da učitavamo podatke iz baze kao i upisujemo podatke u bazu.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package cs102.v12;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

/**
 * Hibernate Utility class with a convenient method to get Session Factory
 * object.
 *
 * @author rados
 */
public class HibernateUtil {

    private static final SessionFactory sessionFactory;

    static {
        try {
            // Create the SessionFactory from standard (hibernate.cfg.xml)
            // config file.
            sessionFactory = new
```

```
AnnotationConfiguration().configure().buildSessionFactory();
        } catch (Throwable ex) {
            // Log the exception.
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
        //        Configuration configuration = new Configuration().configure();
//        StandardServiceRegistryBuilder builder = new
StandardServiceRegistryBuilder().
//                applySettings(configuration.getProperties());
//        sessionFactory = configuration.buildSessionFactory(builder.build());
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }
}
```

# CRUDPREDMET KLASA

## *U zasebnom paketu kreiramo klasu CrudPredmet.java*

Sledeći korak je kreiranje klasa koje obezbeđuju osnovne CRUD operacije ka bazi podataka (CrudProfesor i CrudPredmet). Dakle, za svaku operaciju potrebno je da otvorimo Hibernate sesiju koja nam obezbeđuje razne metode za manipulaciju sa bazom podataka kao što su list, get, save, saveOrUpdate, itd. Takođe je moguće da se transakcija ka bazi podataka vrši metodom beginTransaction(), zatim izvrši modifikacija baze metodom persist i na kraju metodom commit izvrši transakcija.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package cs102.v12;

import java.util.List;
import model.Predmet;
import model.Profesor;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;

/**
 *
 * @author rados
 */
public class CrudPredmet {
```

```
    private static Session session;

    public static List readAll() {
        session = HibernateUtil.getSessionFactory().openSession();
        List predmeti = null;
        try {
            predmeti = session.createQuery("from Predmet").list();
            System.out.println("Uspesno ucitavanje.");
        } catch (HibernateException e) {
            System.out.println("Hibernate exception happend...");
            e.printStackTrace();
        } finally {
            session.close();
        }
        return predmeti;
    }

    public static Predmet read(int id) {
        session = HibernateUtil.getSessionFactory().openSession();
        Predmet predmet = null;
        try {
            predmet = (Predmet) session.get(Predmet.class, new Integer(id));
            System.out.println("Uspesno ucitavanje.");
        } catch (HibernateException e) {
            System.out.println("Hibernate exception happend...");
            e.printStackTrace();
        } finally {
            session.close();
        }
        return predmet;
    }

    public static void insert(Predmet predmet) {
        session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = null;
        try {
            transaction = session.beginTransaction();
            session.persist(predmet);
            //session.save(profesor);
            transaction.commit();
            System.out.println("Uspesan upis u bazu.");
        } catch (HibernateException e) {
            transaction.rollback();
            System.out.println("Hibernate exception happend...");
            e.printStackTrace();
        } finally {
            session.close();
        }
    }

    public static void update(Predmet newPredmet) {
        session = HibernateUtil.getSessionFactory().openSession();
```

```
        Transaction transaction = null;
        try {
            transaction = session.beginTransaction();
            session.saveOrUpdate(newPredmet);
            transaction.commit();
            System.out.println("Uspesno azuriranje.");
        } catch (HibernateException e) {
            transaction.rollback();
            System.out.println("Hibernate exception happend...");
            e.printStackTrace();
        } finally {
            session.close();
        }
    }

    public static void delete(Predmet subToDelete) {
        session = HibernateUtil.getSessionFactory().openSession();
        try {
            session.delete(subToDelete);
            System.out.println("Uspesno brisanje.");
        } catch (HibernateException e) {
            System.out.println("Hibernate exception happend...");
            e.printStackTrace();
        } finally {
            session.close();
        }
    }
}
```

# CRUDPROFESOR KLASA

*U istom paketu gde je i klasa CrudPredmet kreiramo CrudProfesor.java klasu*

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package cs102.v12;

import java.util.List;
import model.Profesor;
import org.hibernate.HibernateException;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;

/**
```

```
 *
 * @author rados
 */
public class CrudProfesor {

    private static Session session;

    public static List<Profesor> readAll() {
        session = HibernateUtil.getSessionFactory().openSession();
        List<Profesor> profesori = null;
        Transaction transaction = null;
        try {
            transaction = session.beginTransaction();
            profesori = session.createQuery("from Profesor").list();
            transaction.commit();
            System.out.println("Uspesno ucitavanje.");
        } catch (HibernateException e) {
            transaction.rollback();
            System.out.println("Hibernate exception happend...");
            e.printStackTrace();
        } finally {
            session.close();
        }
        return profesori;
    }

    public static Profesor read(int id) {
        session = HibernateUtil.getSessionFactory().openSession();
        Profesor profesor = null;
        Transaction transaction = null;
        try {
            transaction = session.beginTransaction();
            profesor = (Profesor) session.get(Profesor.class, new Integer(id));
            transaction.commit();
            System.out.println("Uspesno ucitavanje.");
        } catch (HibernateException e) {
            transaction.rollback();
            System.out.println("Hibernate exception happend...");
            e.printStackTrace();
        } finally {
            session.close();
        }
        return profesor;
    }

    public static void insert(Profesor profesor) {
        session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = null;
        try {
            transaction = session.beginTransaction();
            session.persist(profesor);
            //session.save(profesor);
            transaction.commit();
```

```
            System.out.println("Uspesan upis u bazu.");
        } catch (HibernateException e) {
            transaction.rollback();
            System.out.println("Hibernate exception happend...");
            e.printStackTrace();
        } finally {
            session.close();
        }
    }

    public static void update(int id, Profesor newProf) {
        session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = null;
        try {
            transaction = session.beginTransaction();
            //Profesor prof = read(id);
            // prva varijanta
            // session.saveOrUpdate(newProf);
            // druga varijanta
//            newProf.setId(id);
//            if(prof != null){
//                session.update(newProf);
//            }

            // treca varijanta
            Query query = session.createQuery("from Profesor where id = :id");
            query.setInteger("id", id);
            Profesor prof = (Profesor) query.uniqueResult();
            if (prof != null) {
                newProf.setId(id);
                session.update(newProf);
            }
            transaction.commit();
            System.out.println("Uspesno azuriranje.");
        } catch (HibernateException e) {
            transaction.rollback();
            System.out.println("Hibernate exception happend...");
            e.printStackTrace();
        } finally {
            session.close();
        }
    }

    public static void delete(int id) {
        session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = null;
        try {
            transaction = session.beginTransaction();
            //Profesor prof = read(id);
            // prva varijanta
//            if(prof != null){
//                session.delete(prof);
//            }
```

```
            // druga varijanta
            Query query = session.createQuery("from Profesor where id = :id");
            query.setInteger("id", id);
            Profesor exProf = (Profesor) query.uniqueResult();
            session.delete(exProf);
            transaction.commit();
            System.out.println("Uspesno brisanje.");
        } catch (HibernateException e) {
            transaction.rollback();
            System.out.println("Hibernate exception happend...");
            e.printStackTrace();
        } finally {
            session.close();
        }
    }

    public static void update(Profesor newProf) {
        session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = null;
        try {
            transaction = session.beginTransaction();
            session.saveOrUpdate(newProf);
            transaction.commit();
            System.out.println("Uspesno azuriranje.");
        } catch (HibernateException e) {
            transaction.rollback();
            System.out.println("Hibernate exception happend...");
            e.printStackTrace();
        } finally {
            session.close();
        }
    }

    public static void delete(Profesor profToDelete) {
        session = HibernateUtil.getSessionFactory().openSession();
        Transaction transaction = null;
        try {
            transaction = session.beginTransaction();
            Profesor prof = (Profesor) session.get(Profesor.class,
profToDelete.getId());
            if(prof != null){
                session.delete(prof);
                transaction.commit();
                System.out.println("Uspesno brisanje.");
            } else {
                System.out.println("Profesor se ne nalazi se u bazi.");
            }
        } catch (HibernateException e) {
            transaction.rollback();
            System.out.println("Hibernate exception happend...");
            e.printStackTrace();
        } finally {
            session.close();
```

```
        }
    }

}
```

# VIZUELNI SLOJ APLIKACIJE

*Cilj sekcije da se opiše kreiranje vizuelnog sloja aplikacije*

Što se tiče klasa koje se odnose na GUI komponente u okviru aplikacije one obuhvataju poznavanje JavaFX biblioteke koja je već obrađivana u ranijim lekcijama, tako da će u ovom segmentu biti opisano za šta je koja klasa namenjena.

Klase ProfesorForm i PredmetForm predstavljaju jednostavne JavaFX forme za unos podataka koji se koriste za dodavanje i ažuriranje baze podataka. U okviru ovih klasa je realizovana verifikacija unetih podataka u formu korišćenjem naprednijih data binding mogućnosti u okviru JaveFX.

Klase ProfesorView i PredmetView služe za prikaz sadržaja baze podataka i obuhvataju dugmiće koji omogućavaju brisanje i ažuriranje određenih zapisa iz baze podataka. Klikom na dugme Delete će se obrisati zapis iz baze podataka dok će se klikom na dugme Edit korisniku ponovo otvoriti forma za unos novih podataka.

Klase PredmetRecordView i ProfesorRecordView predstavljaju pomoćne GUI komponente koje u okviru klasa PredmetView i ProfesorView reprezentuju jedan zapis iz baze podataka.

# PREDMETVIEW KLASA

*Naredni zadatak nam je da kreiramo VIEW paket i klasu PredmetView*

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package view;

import cs102.v12.CrudPredmet;
import cs102.v12.CrudProfesor;
import cs102.v12.ProjectSingleton;
import java.util.ArrayList;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import model.Predmet;
```

```
/**
 *
 * @author rados
 */
public class PredmetView extends Stage {
    private VBox root;

    public PredmetView() {
        root = new VBox(10);
        root.setPadding(new Insets(10));
        createList();
        Scene scene = new Scene(root, 450, 400);
        setTitle("Profesori u bazi");
        setScene(scene);
        show();
        ProjectSingleton.getInstance().closeOpenedSecondStage();
        ProjectSingleton.getInstance().setOpenedSecondStage(this);
    }

    public void createList() {
        root.getChildren().clear();
        ArrayList list = (ArrayList) CrudPredmet.readAll();
        for (int i = 0; i < list.size(); i++) {
            root.getChildren().add(new PredmetRecordView(list.get(i)));
        }
    }
}
```

# PROFESORVIEW KLASA

*Naredni zadatak nam je da kreiramo VIEW paket i klasu ProfesorView*

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package view;

import cs102.v12.CrudProfesor;
import cs102.v12.ProjectSingleton;
import java.util.ArrayList;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import model.Profesor;

/**
```

```
 *
 * @author rados
 */
public class ProfesorView extends Stage {

    private VBox root;

    public ProfesorView() {
        root = new VBox(10);
        root.setPadding(new Insets(10));
        createList();
        Scene scene = new Scene(root, 400, 400);
        setTitle("Profesori u bazi");
        setScene(scene);
        show();
        ProjectSingleton.getInstance().closeOpenedSecondStage();
        ProjectSingleton.getInstance().setOpenedSecondStage(this);
    }

    public void createList() {
        root.getChildren().clear();
        ArrayList<Profesor> list = (ArrayList<Profesor>) CrudProfesor.readAll();
        for (int i = 0; i < list.size(); i++) {
            root.getChildren().add(new ProfesorRecordView(list.get(i)));
        }
    }

}
```

# PREDMETFORM KLASA

*U view paketu se nalazi i PredmetForm klasa*

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package view;

import cs102.v12.CrudPredmet;
import cs102.v12.CrudProfesor;
import cs102.v12.ProjectSingleton;
import java.util.ArrayList;
import java.util.List;
import java.util.stream.Collectors;
import javafx.beans.binding.BooleanBinding;
import javafx.beans.property.SimpleBooleanProperty;
import javafx.beans.value.ChangeListener;
```

```
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ChoiceBox;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import javax.swing.JOptionPane;
import model.Predmet;
import model.Profesor;

/**
 *
 * @author rados
 */
public class PredmetForm extends Stage {

    private TextField tfNaziv;
    private TextField tfSifra;
    private TextField tfEspb;
    private ArrayList<Profesor> profesori;
    private ChoiceBox<String> cbProfesor;
    private Button btnAdd;
    private Predmet predmet;
    private Profesor selectedProfesor;
    private SimpleBooleanProperty profSelected;

    public PredmetForm() {
        profSelected = new SimpleBooleanProperty(false);
        setupGUI();
        readProfesori();
        prepareBinding();
        addInsertListener();
        ProjectSingleton.getInstance().closeOpenedSecondStage();
        ProjectSingleton.getInstance().setOpenedSecondStage(this);
    }

    public PredmetForm(Predmet predmet) {
        this.predmet = predmet;
        profSelected = new SimpleBooleanProperty(true);
        setupGUI();
        readProfesori();
        prepareBinding();
        prepareForUpdate();
        addUpdateListener();
        btnAdd.setText("Azuriraj");
        ProjectSingleton.getInstance().closeOpenedSecondStage();
```

```
        ProjectSingleton.getInstance().setOpenedSecondStage(this);
    }

    private void readProfesori() {
        profesori = (ArrayList<Profesor>) CrudProfesor.readAll();
        List<String> profString = profesori.stream().map(prof -> (prof.getId() + "
" + prof.getIme() + " " + prof.getPrezime())).collect(Collectors.toList());
        cbProfesor.setItems(FXCollections.observableArrayList(profString));
    }

    private void setupGUI() {
        GridPane root = new GridPane();
        Text title = new Text("Predmet");
        Text txtNaziv = new Text("Naziv");
        Text txtSifra = new Text("Sifra");
        Text txtEspb = new Text("Espb");
        Text txtProf = new Text("Profesor");
        tfNaziv = new TextField();
        tfSifra = new TextField();
        tfEspb = new TextField();
        cbProfesor = new ChoiceBox<>();
        btnAdd = new Button("Dodaj");
        root.setAlignment(Pos.CENTER);
        root.setHgap(10);
        root.setVgap(10);
        root.add(title, 0, 0);
        root.add(txtNaziv, 0, 1);
        root.add(tfNaziv, 1, 1);
        root.add(txtSifra, 0, 2);
        root.add(tfSifra, 1, 2);
        root.add(txtEspb, 0, 3);
        root.add(tfEspb, 1, 3);
        root.add(txtProf, 0, 4);
        root.add(cbProfesor, 1, 4);
        root.add(btnAdd, 0, 5);

        Scene scene = new Scene(root, 300, 300);
        setTitle("Unos predmeta");
        setScene(scene);
        show();
    }

    public void prepareBinding() {
        BooleanBinding binding = new BooleanBinding() {
            {
                super.bind(tfNaziv.textProperty(),
                        tfSifra.textProperty(),
                        tfEspb.textProperty(),
                        profSelected);
            }

            @Override
            protected boolean computeValue() {
```

```
                return tfNaziv.getText().isEmpty()
                        || tfSifra.getText().isEmpty()
                        || tfEspb.getText().isEmpty()
                        || !profSelected.get();
            }
        };
        btnAdd.disableProperty().bind(binding);
    }

    private void addInsertListener() {
        btnAdd.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                try {
                    Integer espb = Integer.parseInt(tfEspb.getText());
                    predmet = new Predmet(tfSifra.getText(), tfNaziv.getText(),
espb);
                    predmet.setProfesorId(selectedProfesor);
                    CrudPredmet.insert(predmet);
                } catch (NumberFormatException e) {
                    JOptionPane.showMessageDialog(null, "ESPB bodovi moraju biti
ceo broj.");
                }
            }
        });

        cbProfesor.getSelectionModel().selectedIndexProperty().addListener(new
ChangeListener<Number>() {
            @Override
            public void changed(ObservableValue<? extends Number> observable,
Number oldValue, Number newValue) {
                profSelected.set(true);
                selectedProfesor = profesori.get((int) newValue);
            }
        });
    }

    private void prepareForUpdate() {
        tfSifra.setText(predmet.getSifra());
        tfNaziv.setText(predmet.getNaziv());
        tfEspb.setText(predmet.getEspb() + "");

cbProfesor.getSelectionModel().select(profesori.indexOf(predmet.getProfesorId()));
    }

    private void addUpdateListener() {
        btnAdd.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                try {
                    Integer espb = Integer.parseInt(tfEspb.getText());
                    predmet.setNaziv(tfNaziv.getText());
                    predmet.setSifra(tfSifra.getText());
```

```
                    predmet.setEspb(espb);
                    predmet.setProfesorId(selectedProfesor);
                    CrudPredmet.update(predmet);
                } catch (NumberFormatException e) {
                    JOptionPane.showMessageDialog(null, "ESPB bodovi moraju biti
ceo broj.");
                }
            }
        });

        cbProfesor.getSelectionModel().selectedIndexProperty().addListener(new
ChangeListener<Number>() {
            @Override
            public void changed(ObservableValue<? extends Number> observable,
Number oldValue, Number newValue) {
                profSelected.set(true);
                selectedProfesor = profesori.get((int) newValue);
            }
        });
    }

}
```

# PROFESORFORM KLASA

*U view paketu se nalazi i ProfesorForm klasa*

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package view;

import cs102.v12.CrudProfesor;
import cs102.v12.ProjectSingleton;
import javafx.beans.binding.BooleanBinding;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.layout.GridPane;
import javafx.scene.text.Text;
import javafx.stage.Stage;
import model.Profesor;

/**
```

```
 *
 * @author rados
 */
public class ProfesorForm extends Stage {

    private TextField tfIme;
    private TextField tfPrezime;
    private TextField tfZvanje;
    private Button btnAdd;
    private Profesor profesor = null;

    public ProfesorForm() {
        setupGUI();
        prepareBinding();
        addInsertListener();
        ProjectSingleton.getInstance().closeOpenedSecondStage();
        ProjectSingleton.getInstance().setOpenedSecondStage(this);
    }

    public ProfesorForm(Profesor prof) {
        this.profesor = prof;
        setupGUI();
        prepareBinding();
        prepareFormForUpdate();
        addUpdateListener();
        btnAdd.setText("Azuriraj");
        ProjectSingleton.getInstance().closeOpenedSecondStage();
        ProjectSingleton.getInstance().setOpenedSecondStage(this);
    }

    private void setupGUI() {
        GridPane root = new GridPane();
        Text title = new Text("Profesor");
        Text txtIme = new Text("Ime");
        Text txtPrezime = new Text("Prezime");
        Text txtZvanje = new Text("Zvanje");
        tfIme = new TextField();
        tfPrezime = new TextField();
        tfZvanje = new TextField();
        btnAdd = new Button("Dodaj");
        root.setAlignment(Pos.CENTER);
        root.setHgap(10);
        root.setVgap(10);
        // dodavanje kontrola
        root.add(title, 0, 0);
        root.add(txtIme, 0, 1);
        root.add(tfIme, 1, 1);
        root.add(txtPrezime, 0, 2);
        root.add(tfPrezime, 1, 2);
        root.add(txtZvanje, 0, 3);
        root.add(tfZvanje, 1, 3);
        root.add(btnAdd, 0, 4);
```

```
        Scene scene = new Scene(root, 300, 250);
        setTitle("Unos profesora");
        setScene(scene);
        show();
    }

    public void prepareBinding(){
        BooleanBinding binding = new BooleanBinding() {
            {
                super.bind(tfIme.textProperty(),
                        tfPrezime.textProperty(),
                        tfZvanje.textProperty());
            }

            @Override
            protected boolean computeValue() {
                return tfIme.getText().isEmpty()
                        || tfPrezime.getText().isEmpty()
                        || tfZvanje.getText().isEmpty();
            }
        };
        btnAdd.disableProperty().bind(binding);
    }

    private void addInsertListener() {

        btnAdd.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                profesor = new Profesor(tfIme.getText(), tfPrezime.getText(),
tfZvanje.getText());
                CrudProfesor.insert(profesor);
            }
        });
    }


    private void addUpdateListener() {

        btnAdd.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                profesor.setIme(tfIme.getText());
                profesor.setPrezime(tfPrezime.getText());
                profesor.setZvanje(tfZvanje.getText());
                CrudProfesor.update(profesor);
            }
        });
    }

    private void prepareFormForUpdate(){
        tfIme.setText(profesor.getIme());
        tfPrezime.setText(profesor.getPrezime());
```

```
        tfZvanje.setText(profesor.getZvanje());
    }

}
```

# PREDMETRECORDVIEW KLASA

*U view paketu se nalazi i PredmetRecordView klasa*

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package view;

import cs102.v12.CrudPredmet;
import cs102.v12.ProjectSingleton;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.scene.text.Text;
import model.Predmet;

/**
 *
 * @author rados
 */
public class PredmetRecordView extends HBox {

    private Predmet recordObj;
    private Text txtRecord;
    private Button btnEdit;
    private Button btnDelete;

    public PredmetRecordView(Predmet subject) {
        this.recordObj = subject;
        setSpacing(10);
        btnEdit = new Button("Edit");
        btnDelete = new Button("Delete");
        txtRecord = new Text(recordObj.toString());
        txtRecord.setWrappingWidth(250);
        btnDelete.setPrefWidth(60);
        btnEdit.setPrefWidth(60);
        addListeners();
        getChildren().addAll(txtRecord, btnEdit, btnDelete);
    }
```

```
    public void addListeners() {
        btnEdit.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                new PredmetForm(recordObj);
            }
        });

        btnDelete.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                CrudPredmet.delete(recordObj);
                PredmetView subView = (PredmetView)
ProjectSingleton.getInstance().getOpenedSecondStage();
                subView.createList();
            }
        });
    }

}
```

# PROFESORRECORD VIEW KLASA

*U view paketu se nalazi i ProfesorRecord klasa*

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package view;

import cs102.v12.CrudProfesor;
import cs102.v12.ProjectSingleton;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.control.Button;
import javafx.scene.layout.HBox;
import javafx.scene.text.Text;
import model.Profesor;

/**
 *
 * @author rados
 */
public class ProfesorRecordView extends HBox {

    private Profesor recordObj;
    private Text txtRecord;
```

```
    private Button btnEdit;
    private Button btnDelete;

    public ProfesorRecordView(Profesor prof) {
        this.recordObj = prof;
        setSpacing(10);
        btnEdit = new Button("Edit");
        btnDelete = new Button("Delete");
        txtRecord = new Text(recordObj.toString());
        txtRecord.setWrappingWidth(220);
        btnDelete.setPrefWidth(60);
        btnEdit.setPrefWidth(60);
        addListeners();
        getChildren().addAll(txtRecord, btnEdit, btnDelete);
    }

    public void addListeners() {
        btnEdit.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                new ProfesorForm(recordObj);
            }
        });

        btnDelete.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                CrudProfesor.delete(recordObj);
                ProfesorView profView = (ProfesorView)
ProjectSingleton.getInstance().getOpenedSecondStage();
                profView.createList();
            }
        });
    }

}
```

# PROJECTSINGLETON.JAVA

## *U istom paketu gde su i crud klase, kao i main klasa kreirati sledeću klasu*

Klasa ProjectSingleton je klasa koja kreira samo jednu instancu i u ovoj aplikaciji je zadužena da obezbedi da u jednom trenutku možemo da pokrenemo maksimalno dva prozora. Ova klasa može da sadrži bilo koje parametre koji su globalni na nivou celog projekta i kojima se treba pristupati iz različitih klasa bez kreiranja posebne instance. Dakle, postoji samo jedna instanca i stanje te instance treba da bude dostupno svim ostalim klasama u projektu. Ukoliko neka klasa utiče na promenu stanja singleton klase, te promene će biti dostupne svim ostalim klasama. Ovo je samo jedna od primena signleton klase.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package cs102.v12;

import javafx.stage.Stage;

/**
 *
 * @author rados
 */
public class ProjectSingleton {
    private static ProjectSingleton instance;
    private Stage openedSecondStage = null;

    private ProjectSingleton() {
    }

    public static ProjectSingleton getInstance() {
        if(instance == null){
            instance = new ProjectSingleton();
        }
        return instance;
    }

    public void closeOpenedSecondStage() {
        if(openedSecondStage != null){
            openedSecondStage.close();
        }
    }

    public Stage getOpenedSecondStage() {
        return openedSecondStage;
    }

    public void setOpenedSecondStage(Stage openedSecondStage) {
        this.openedSecondStage = openedSecondStage;
    }

}
```

# MAIN KLASA

## *Kreiramo Aplikaciju*

Klasa Main je pokretačka JavaFX klasa i u okviru nje je realizovan početni meni aplikacije.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package cs102.v12;

import javafx.application.Application;
import javafx.beans.property.SimpleBooleanProperty;
import javafx.beans.value.ChangeListener;
import javafx.beans.value.ObservableValue;
import javafx.collections.FXCollections;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.ComboBox;
import javafx.scene.layout.VBox;
import javafx.stage.Stage;
import javafx.stage.WindowEvent;
import view.PredmetForm;
import view.PredmetView;
import view.ProfesorForm;
import view.ProfesorView;

/**
 *
 * @author rados
 */
public class Main extends Application {

    private ComboBox<String> tableChoice;
    private Button btnNew;
    private Button btnRead;
    private SimpleBooleanProperty selected;
    private String selectedTable = "";

    @Override
    public void start(Stage primaryStage) {
        VBox root = new VBox();
        selected = new SimpleBooleanProperty(false);
        root.setSpacing(20);
        root.setAlignment(Pos.CENTER);
        tableChoice = new ComboBox<>(FXCollections.observableArrayList("Profesori",
"Predmeti"));
        btnNew = new Button("Upis u bazu");
        btnRead = new Button("Prikaz sadrzaja iz baze");

        root.getChildren().addAll(tableChoice, btnNew, btnRead);
        Scene scene = new Scene(root, 300, 250);

        addListeners();
```

```
        primaryStage.setTitle("Hibernate demo!");
        primaryStage.setScene(scene);
        primaryStage.show();

        primaryStage.setOnCloseRequest(new EventHandler<WindowEvent>() {
            @Override
            public void handle(WindowEvent event) {
                System.exit(0);
            }
        });
    }

    private void addListeners() {
        btnNew.disableProperty().bind(selected.not());
        btnRead.disableProperty().bind(selected.not());
        tableChoice.valueProperty().addListener(new ChangeListener<String>() {
            @Override
            public void changed(ObservableValue<? extends String> observable,
String oldValue, String newValue) {
                selected.set(true);
                selectedTable = newValue;
            }
        });

        EventHandler<ActionEvent> newHandler = new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                if (selectedTable.equalsIgnoreCase("Profesori")) {
                    new ProfesorForm();
                }
                if (selectedTable.equalsIgnoreCase("Predmeti")) {
                    new PredmetForm();
                }
            }
        };

        EventHandler<ActionEvent> readHandler = new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent event) {
                if (selectedTable.equalsIgnoreCase("Profesori")) {
                    new ProfesorView();
                }
                if (selectedTable.equalsIgnoreCase("Predmeti")) {
                    new PredmetView();
                }
            }
        };

        btnNew.setOnAction(newHandler);
        btnRead.setOnAction(readHandler);
    }

    /**
```

```
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        launch(args);
    }

}
```

# DEMONSTRACIJA PROGRAMA

*Cilje sekcije je da se prikažu prozori u okviru aplikacije.*

Početni meni:

Slika 10.9 Prikaz početnog menija aplikacije

Forma za dodavanje profesora:

Forma i prikaz predmeta:

Slika 10.10 Forma za unos predmeta

Slika 10.11 Prikaz predmeta iz baze

# Poglavlje 11

# Domaći zadatak

## DOMAĆI ZADACI

*Za ove zadatke se ne daje rešenje i očekuje se da svaki student pokuša samostalno rešavanje istih*

**Zadatak 1**

Za klasu koju ste odabrali u prethodnom domaćem zadatku napraviti JavaFX aplikaciju koja koristi Hibernate framework za rad sa bazom podataka. Potrebno izgenerisati odgovarajuću klasu na osnovu entiteta iz baze, kreirati HibernateUtil klasu, kao i hibernate konfiguracionu xml datoteku. Nakon toga kreirati posebnu statičku ili Singleton klasu(po izboru) koja omogućava CRUD operacije nad bazom. Na kraju korišćenjem JavaFX kontrola kreirati forme koje će omogućiti unos podataka u bazu, brisanje, izmenu i prikaz svih podataka iz baze.

# Poglavlje 12

# Zaključak

## ZAKLJUČAK

### *Završni rezime*

U ovoj lekciji se govorilo o osnovama Hiberate -a, njegovoj arhitekturi, i idejama iz kojih je nastao. Problem i tehnike ovog alata su analizirane dovoljno za osnovne potrebe za rad sa Hiberate-om. Hibernate je postao najpopularnij ORM za trajno uskladitenje objekata u relacione baze podataka, kao i njihovo prikupljanje iz baze radi korišćenja u objektno-orijentisanoj aplikaciji.

## REFERENCE

### *Literatura korišćena u ovoj lekciji*

1.. Hibernate Tutorial, Tutorialspoint, http://www.tutorialspoint.com/hibernate/index.htm