2. **Hibernate Architecture**

Hibernate architecture ထဲမွာ persistent object, session factory, transaction factory, connection factory, session, transaction etc လိုမ်ိဳး object ေပါင္းမ်ားစြာ ပါ၀င္ပါတယ္။

Hibernate ကို layer ၄ ခု ခြဲထားပါတယ္။

1. Java application layer
2. Hibernate framework layer
3. Backhand api layer
4. Database layer



ဒီပံုကေတာ့ hibernate ကို mapping file ရယ္ configuration file ေတြနဲ႔ ေတြ႕ၿမင္ရတဲ့ high level architecture ပံုပဲ ၿဖစ္ပါတယ္။



hibernate framework က session factory, session, transaction etc တို႔လို object မ်ားစြာကို java ရဲ႕ ရွိၿပီးသား api မ်ားၿဖစ္ၾကတဲ့ DBC (Java Database Connectivity), JTA (Java Transaction API) and JNDI (Java Naming Directory Interface)တို႔နဲ႔ပါင္းစပ္အသံုးၿပဳတယ္။

Hibernate Architecture ရဲ႕ elements မ်ား

SessionFactory

session တစ္ခုရဲ႕ factory တစ္ခုၿဖစ္သလို ConnectionProvider ရဲ႕ client လည္းၿဖစ္တယ္။ သူက data ရဲ႕ second level cache (optional) ကို hold လုပ္ထားတယ္။ org.hibernate.SessionFactory interface က session ရဲ႕ object ကို ရရွိဖို႔ရန္ factory method ကို ထုတ္ေပးပါတယ္။

Session

session object က application နဲ႔ database ထဲမွာ သိမ္းထားတဲ့ data နဲ႔ၾကားထဲမွာ interface တစ္ခုကို ထုတ္ေပးပါတယ္။ သူက short-lived object တစ္ခုၿဖစ္ၿပီးေတာ့ JDBC object ကို wrap လုပ္ေပးပါတယ္။ session က Transaction, Query နဲ႔ Criteria တို႔ရဲ႕ factory လည္း ဟုတ္တယ္။ session က data ရဲ႕ first-level cache (mandatory) ကို hold လုပ္ထားတယ္။ org.hibernate.Session interface က object ေတြ insert, update နဲ႔ delete လုပ္ဖို႔ရန္ method မ်ားကို ထုတ္ေပးပါတယ္။ ၿပီးေတာ့ Transaction, Query နဲ႔ Criteria အတြက္လည္း factory method မ်ားကို ထုတ္ေပးပါတယ္။

Transaction

transaction object က work ရဲ႕ atomic unit ကို သတ္မွတ္ေပးတယ္။ org.hibernate.Transaction interface က transaction management အတြက္ methods မ်ားကို ထုတ္ေပးပါတယ္။

ConnectionProvider

JDBC connections ရဲ႕ factory တစ္ခုၿဖစ္တယ္။ connectionProvider က DriverManager သို႔မဟုတ္ DataSource မွ application ကို abstract လုပ္တယ္။ သူကေတာ့ optional ၿဖစ္တယ္။

TransactionFactory

Transaction ရဲ႕ factory တစ္ခုၿဖစ္တယ္။ သူကလည္း optional ၿဖစ္တယ္။

**Hibernate with Eclipse**

1. HB Using XML (main method နဲ႔ run ရမယ္)

2. HB Using Annotation (main method နဲ႔ run ရမယ္)

**1. HB Using XML**

Hibernate Example using XML in Eclipse

ေအာက္ပါအဆင့္မ်ားကို လုပ္ေဆာင္ရမွာ ၿဖစ္ပါတယ္။

1. persistence class ကို create လုပ္မယ္။
2. persistence class အတြက္ mapping file ကို create လုပ္မယ္။
3. configuration file ကို create လုပ္မယ္။
4. table ေဆာက္မယ္။
5. persistent object ကို retrieves သို႔မဟုတ္ stores လုပ္မယ့္ class တစ္ခုကို create လုပ္မယ္။
6. jar file ကို load လုပ္မယ္။
7. main method ပါတဲ့ class ကို run မယ္။

1. Create the Persistent class

simple Persistent class တစ္ခုကို create လုပ္ေတာ့မယ္ဆိုရင္ ေအာက္ပါ rules မ်ားကို လိုက္နာေဆာင္ရြက္သင့္ပါတယ္။

* **A no-arg constructor**: default constructor တစ္ခုကေတာ့ ပါကိုပါရမယ္ ဒါမွသာ hibernate က Persistent class ရဲ႕ instance ကို newInstance() method နဲ႔ ေဆာက္လို႔ရမွာ ၿဖစ္ပါတယ္။
* **identifier property တစ္ခုေပးပါ** : အဲ့တာက id လို attribute မ်ိဳးကို assign လုပ္လို႔ ပိုမိုေကာင္းမြန္ေစပါတယ္။ id ဆိုတဲ့ field ကေတာ့ db ထဲမွာ primary key ပံုစံနဲ႔ ရွိေနမွာ ၿဖစ္ပါတယ္။
* **getter and setter methods မ်ားထည့္သြင္းပါ** : Hibernate က default အားၿဖင့္ getter setter name မ်ားကို အသိအမွတ္ၿပဳထားေပးၿပီးသား။
* **non-final class ကို ဦးစားေပးပါ** : hibernate က proxies ရဲ႕ concept ကို အသံုးၿပဳတယ္ ဆိုလိုသည္မွာ အဲ့တာက persistence class အေပၚမွာ depends ၿဖစ္ေနပါတယ္။ application programmer ေတြကေတာ့ lazy association fetching အတြက္ proxies မ်ားကို သံုးဖို႔ မၿဖစ္ႏိုင္ပါဘူး။

Employee.java

**public** **class** Employee {

**private** **int** id;

**private** String firstName, lastName;

**getter//setter**

2. Persistent class အတြက္ mapping file ကို create လုပ္ၿခင္း

mapping file ရဲ႕ နာမည္ကို အစဥ္အလာအတိုင္းပဲ ေပးသင့္ပါတယ္။ **class\_name.hbm.xml** အဲ့လို format အတိုင္းပဲ ၿဖစ္သင့္တယ္။

* **hibernate-mapping** : mapping file ရဲ႕ root element တစ္ခုၿဖစ္တယ္။ သူ႕ထဲမွာ mapping file အားလံုးပါ၀င္ပါတယ္။
* **class** : hibernate-mapping element ရဲ႕ sub element ၿဖစ္တယ္။ သူက Persistent class ကို သတ္မွတ္ေပးပါတယ္။
* **generator** : id ရဲ႕ sub element တစ္ခုၿဖစ္တယ္။ သူ႕ကို primary key generate လုပ္ဖို႔ရန္ အသံုးၿပဳတယ္။ သူထဲမွာ assigned, increment, hilo, sequence, native etc လိုမ်ိဳး generator class မ်ားစြာ ပါ၀င္ႏိုင္ပါတယ္။
* **property** : class ရဲ႕ sub element တစ္ခုၿဖစ္ပါတယ္။ သူက persistence class ရဲ႕ property name ကို သတ္မွတ္ေပးပါတယ္။

သူ႕ကို eclipse မွာ ယူမယ္ဆိုရင္ xml file type ကို မယူပါ။ file ဆိုတဲ့ type ကို ယူၿပီးမွာ name ေပးတဲ့အခါမွ format ၿဖစ္တဲ့ xml ကို ထည့္ရိုက္ေပးလိုက္တာ ၿဖစ္ပါတယ္။

ၿပီးေတာ့ သူ႕ကို local package ၿဖစ္တဲ့ hb ရဲ႕ အၿပင္ဘက္မွာ ေဆာက္ေပးပါ။ ရွင္းရွင္းေၿပာေၿပာရရင္ java file ေတြက local package အတြင္းထဲမွာ ၿဖစ္ၿပီးေတာ့ xml format မ်ားကေတာ့ local package ရဲ႕ အၿပင္ဘက္မွာ ၿဖစ္ရမွာ ၿဖစ္ပါတယ္။

employee.hbm.xml

<hibernate-mapping>

<class name=*"hibernate\_with\_eclipse.hb\_with\_xml.hb.Employee"* table=*"emp1000"*>

<id name=*"id"*>

<generator class=*"assigned"*></generator>

</id>

<property name=*"firstName"*></property>

<property name=*"lastName"*></property>

</class>

</hibernate-mapping>

3. Configuration file ကို create လုပ္ၿခင္း

mapping file နဲ႔ database နဲ႔ပတ္သက္တဲ့ information မ်ားပါ၀င္ပါတယ္။ ဥပမာ connection\_url, driver\_class, username, password etc. တို႔ၿဖစ္ၾကပါတယ္။database ထဲမွာ table ကို automatically create လုပ္ဖို႔ရန္ hbm2ddl.auto property ကို အသံုးၿပဳပါတယ္။ Dialect class ရဲ႕ detail အေၾကာင္းကိုေတာ့ next topic မွာ ေၿပာပါမယ္။ သူ႕နာမည္ကေတာ့ hibernate.cfg.xml ၿဖစ္ရပါမယ္။သူ႕ကိုေတာ့ soucrce folder တစ္ခုေဆာက္ၿပီး ထည့္လိုက္ပါတယ္။ xml နဲ႔ေရးထားတဲ့အတြက္ေၾကာင့္ mapping resource ၿဖစ္ေနတာပါ။

hibernate.cfg.xml

<hibernate-configuration>

<session-factory>

<property name=*"hbm2ddl.auto"*>update</property>

<property name=*"dialect"*> org.hibernate.dialect.MySQLDialect</property>

<property name=*"connection.url"*>jdbc:mysql://localhost/hibernate</property>

<property name=*"connection.username"*>root</property>

<property name=*"connection.password"*>root</property>

<property name=*"connection.driver\_class"*>com.mysql.jdbc.Driver</property>

<mapping resource=*"hibernate\_with\_eclipse/hb\_with\_xml/employee.hbm.xml"*/>

</session-factory>

</hibernate-configuration>

5. db မွာ သက္ဆိုင္ရာ table name အတိုင္း ေဆာက္မယ္။

CREATE TABLE `hibernate`.`emp1000` (

`id` INT NOT NULL,

`firstName` VARCHAR(45) NULL,

`secondName` VARCHAR(45) NULL,

PRIMARY KEY (`id`));

6. class တစ္ခုေဆာက္ၿပီး အဲ့ class ထဲကို objects မ်ား retrieves သို႔မဟုတ္ stores လုပ္ၿခင္း

StoreData.java (main method ပါ၀င္ပါတယ္)

**public** **class** StoreData {

**public** **static** **void** main(String[] args) {

StandardServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();

Metadata meta = **new** MetadataSources(ssr).getMetadataBuilder().build();

SessionFactory factory = meta.getSessionFactoryBuilder().build();

Session session = factory.openSession();

Transaction t = session.beginTransaction();

Employee e1 = **new** Employee();

e1.setId(1);

e1.setFirstName("Gaurav");

e1.setLastName("Chawla");

session.save(e1);

t.commit();

System.***out***.println("successfully saved");

factory.close();

session.close();}}

7. hibernate အတြက္ လိုအပ္ေသာ jar file မ်ားကို download ဆြဲၿခင္း နဲ႔ eclipse ထဲထည့္သြင္းၿခင္း

လိုအပ္ေသာ jar file မ်ားကို download ဆြဲၿပီး resource folder ေအာက္မွာ jar ဆိုတဲ့ folder တစ္ခုေဆာက္ၿပီး ထည့္ေပးလိုက္တယ္။ ၿပီးရင္ build path မွာ add external jar ကို click ႏွိပ္လိုက္ရင္ folder dialog တက္လာတယ္ အဲ့တာဆိုရင္ ခုနက ကိုယ္ resource ေအာက္မွာထည့္ထားတဲ့ folder path လမ္းေၾကာင္းေအာက္က jar file ေတြကုို ေရြးေပးလိုက္ရံုပဲ။

**2. HB using Annotation**

hibernate application ကို xml နဲ႔ create လုပ္လို႔ရလို annotation နဲ႔လည္း create လုပ္လို႔ရပါတယ္။ ဥပမာ @Entity, @Id, @Table etc တို႔ ၿဖစ္ၾကပါတယ္။ Hibernate Annotations က JPA 2 specification ကို အေၿခခံထားတာၿဖစ္ၿပီး features ေတြအားလံုးကိုလည္း

support လုပ္ေပးထားပါတယ္။ JAP annotations အားလံုးကို javax.persistence package ထဲမွာ သတ္မွတ္ထားတာ ၿဖစ္ပါတယ္။ Hibernate EntityManager က JPA specification က သတ္မွတ္ထားတဲ့ interfaces နဲ႔ life cycle ကို implements လုပ္ထားပါတယ္။ annotation နဲ႔ hibernate ကို develop လုပ္ၿခင္းက ဘာေကာင္းလဲဆိုေတာ့ mapping (hbm) file ကို create လုပ္စရာမလိုေတာ့ပါ။ hibernate annotations က meta data ကို ေပးႏိုင္ရန္ လုပ္ေဆာင္ပါတယ္။

1. Create persistence class

@Entity annotation - ဒီ class က entity ၿဖစ္တဲ့ဆိုတာကို သိသာေစဖို႔အတြက္ ၿဖစ္တယ္။

@Table annotation – entity ထဲက data ေတြကို ဘယ္ table ထဲကို persist လုပ္ရမလဲဆိုတာက ေၿပာၿခင္းၿဖစ္ပါတယ္။ @Table annotation ဆိုတဲ့ annotation ကို အသံုးမၿပဳခဲ့ဘူးဆိုရင္ေတာ့ class name နဲ႔တူတဲ့ table name ကို default အားၿဖင့္ သြားၿပီး persist လုပ္ေပးလိမ့္မယ္။

Student.java

@Entity

@Table(name = "emp500")

**public** **class** Student {

@Id

**private** **int** id;

**private** String firstName, lastName;

**getter//setter**

2. Create the Configuration file

annotation နဲ႔ေရးထားတဲ့အတြက္ေၾကာင့္ mapping class ၿဖစ္ေနတာပါ။

hibernate.cfg.xml

<hibernate-configuration>

<session-factory>

<property name=*"hbm2ddl.auto"*>update</property>

<property name=*"dialect"*> org.hibernate.dialect.MySQLDialect</property>

<property name=*"connection.url"*>jdbc:mysql://localhost/hibernate</property>

<property name=*"connection.username"*>root</property>

<property name=*"connection.password"*>root</property>

<property name=*"connection.driver\_class"*>com.mysql.jdbc.Driver</property>

<mapping class=*"hibernate\_with\_eclipse.hb\_with\_annotation.Student"*/>

<mapping resource=*"hibernate\_with\_eclipse/hb\_with\_xml/employee.hbm.xml"*/>

</session-factory>

</hibernate-configuration>

3. db မွာ သက္ဆိုင္ရာ table name အတိုင္း ေဆာက္မယ္။

CREATE TABLE `hibernate`.`emp500` (

`id` INT NOT NULL,

`firstName` VARCHAR(45) NULL,

`lastName` VARCHAR(45) NULL,

PRIMARY KEY (`id`));

4. Create the class that retrieves or stores the persistent object.

StoredDataAnnotation.java

**public** **class** StoredDataAnnotation {

**public** **static** **void** main(String[] args) {

StandardServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();

Metadata meta = **new** MetadataSources(ssr).getMetadataBuilder().build();

SessionFactory factory = meta.getSessionFactoryBuilder().build();

Session session = factory.openSession();

Transaction t = session.beginTransaction();

Student e1 = **new** Student();

e1.setId(101);

e1.setFirstName("Gaurav");

e1.setLastName("Chawla");

session.save(e1);

t.commit();

System.***out***.println("successfully saved");

factory.close();

session.close();}}

Hibernate Example

1. HB Web Application

2. HB Generator Class

3. HB Dialects

1. **HB Web Application**

Web Application with Hibernate (using XML)

presentation logic အတြက္ jsp ကို သံုးမယ္။ representing data အတြက္ Bean class ကို သံုးမယ္။ database codes အတြက္ DAO class ကုိ သံုးမယ္။ အခု example မွာဆိုရင္ JSP file ကိုသံုးၿပီး user ဆီက data ကို ရယူပါမယ္။

index.jsp

user ဆီက data ယူမယ္ ၿပီးရင္ register.jsp ဆီကို post method နဲ႔ သြားမယ္။

<form action=*"pages/hibernate\_example/hb\_web\_application/register.jsp"* method=*"post"*>

Name:<input type=*"text"* name=*"name"*/><br><br/>

Password:<input type=*"password"* name=*"password"*/><br><br/>

Email ID:<input type=*"text"* name=*"email"*/><br><br/>

<input type=*"submit"* value=*"register"*/>

</form>

register.jsp

register.jsp file ကို user class ကေနလာတဲ့ object ထဲက information ေတြကို သိမ္းထားဖို႔နဲ႔ request parameters အကုန္လံုးကို ရယူဖို႔ရအတြက္ ေရးသားထားၿခင္းၿဖစ္ပါတယ္။ သူက user entity class object ကို userDao ရဲ႕ register method ေခၚၿပီး pass လုပ္ထားပါတယ္။ register.jsp က process ကို အဓိကခ်ိတ္ဆက္ေပးတဲ့ file တစ္ခုၿဖစ္ပါတယ္။

<%@page import=*"hibernate\_example.hb\_web\_application.UserDao"*%>

<jsp:useBean id=*"obj"* class=*"hibernate\_example.hb\_web\_application.User"*>

</jsp:useBean>

<jsp:setProperty property=*"\*"* name=*"obj"* />

<%

**int** i = UserDao.register(obj);

**if** (i > 0)

out.print("You are successfully registered");

%>

User.java

**public** **class** User {

**private** **int** id;

**private** String name, password, email;

**public** User() {}

//getter and setter

user.hbm.xml

database ထဲက table နဲ႔ user entity နဲ႔ map လုပ္ေပးတဲ့ file ၿဖစ္ပါတယ္။

<hibernate-mapping>

<class name=*"hibernate\_example.hb\_web\_application.User"* table=*"u400"*>

<id name=*"id"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"name"*></property>

<property name=*"password"*></property>

<property name=*"email"*></property>

</class>

</hibernate-mapping>

UserDao.java

parameter ပါလာတဲ့ user object ကို သိမ္းထားေပးၿပီး register page ကို return ၿပန္ေပးလိုက္တယ္။

**public** **class** UserDao {

**public** **static** **int** register(User u) {

**int** i = 0;

StandardServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();

Metadata meta = **new** MetadataSources(ssr).getMetadataBuilder().build();

SessionFactory factory = meta.getSessionFactoryBuilder().build();

Session session = factory.openSession();

Transaction t = session.beginTransaction();

i = (Integer) session.save(u);

t.commit();

session.close();

**return** i;}}

hibernate.cfg.xml

database information နဲ႔ mapping file information ကို သိမ္းဆည္းေပးပါတယ္။

<mapping resource=*"hibernate\_example/hb\_web\_application/user.hbm.xml"*/>

web.xml

<welcome-file>pages/hibernate\_example/hb\_web\_application/index.jsp</welcome-file>

2. HB Generator Class

<generator> class က id ရဲ႕ sub element တစ္ခုၿဖစ္တယ္။ persistent class ရဲ႕ object ေတြအတြက္ unique identifie ကို ထုတ္ေပးဖို႔ရန္ အသံုးၿပဳပါတယ္။ Hibernate Framework ထဲမွာ generator class မ်ားစြာ သတ္မွတ္ေပးထားပါတယ္။

generator class အားလံုးက org.hibernate.id.IdentifierGenerator interface ကို implements လုပ္ထားပါတယ္။ application programme ေတြက IdentifierGenerator interface ကို implementing လုပ္ၿခင္းအားၿဖင့္ one's own generator classes ေတြကို create လုပ္တယ္။ Hibernate framework ေတြက built-in generator classes ေၿမာက္မ်ားစြာကို ထုတ္ထားေပးၿပီးသား ၿဖစ္ပါတယ္။

1. assigned
2. increment
3. sequence
4. hilo
5. native
6. identity
7. seqhilo
8. uuid
9. guid
10. select
11. foreign
12. sequence-identity

1) assigned

<generator> element မရွိခဲ့ဘူးဆိုရင္ ဒီေကာင္က default အေနနဲ႔၀င္ေနမွာ ၿဖစ္ပါတယ္။

....

 <hibernate-mapping>

  <**class** ...>

    <id ...>

     <generator **class**="assigned"></generator>

    </id>

    .....

  </**class**>

 </hibernate-mapping>

2) increment

table တစ္လံုးထဲကို data ထည့္လိုက္ရင္ id မ်ား into တိုးလာေအာင္ လုပ္ေဆာင္တဲ့ class ၿဖစ္ပါတယ္။ အဲ့ class က short, int နဲ႔ long မ်ိဳးစံုေသာ data type မ်ားကို ထုတ္ေပးႏိုင္ပါတယ္။ အကယ္လို႔မ်ား table မွာ identifier ပါခဲ့မယ္ဆိုရင္ေတာ့ application အေနနဲ႔ maximum value ကို စဥ္းစားေပးဖို႔ လိုလိမ့္မယ္ အဲ့လိုမွမဟုတ္ဘူးဆိုရင္ေတာ့ application အေနနဲ႔ first generated identifier ကို 1 ကေနစၿပီး ထုတ္ေပးပါလိမ့္မယ္။

....

 <hibernate-mapping>

  <**class** ...>

    <id ...>

     <generator **class**="increment"></generator>

    </id>

    .....

  </**class**>

 </hibernate-mapping>

3) Sequence

database ရဲ႕ sequence ကို အသံုးၿပဳပါတယ္။ အကယ္လို႔ sequence ကို မသတ္မွတ္ထားရေသးဘူးဆိုရင္ auto သတ္မွတ္ေပးပါလိမ့္မယ္။ ဥပမာ oracle database မွာဆိုရင္ HIBERNATE\_SEQUENCE ဆိုတဲ့ sequence ကို auto သတ္မွတ္ေပးထားတယ္။ Oracle, DB2, SAP DB, Postgre SQL or McKoi တို႔မွာ sequence ကို သံုးတယ္ သို႔ေပမယ့္ interbase ထဲမွာရွိတဲ့ generator ကို အသံုးၿပဳပါတယ္။

.....

<id ...>

<generator class="sequence"></generator>

</id>

.....

ကိုယ္ပိုင္ sequence တစ္ခု။ generator ရဲ႕ param subelement ကို အသံုးၿပဳထားပါတယ္။

.....

<id ...>

<generator class="sequence">

<param name="sequence">your\_sequence\_name</param>

</generator>

</id>

.....

4) Hilo

short, int နဲ႔ long အစရွိတဲ့ type ရဲ႕ id မ်ားကို ထုတ္ရန္ high နဲ႔ low algorithm မ်ားကို အသံုးၿပဳပါတယ္။

.....

 <id ...>

  <generator **class**="hilo"></generator>

 </id>

 .....

5) native

database vendor ေပၚမူတည္ၿပီး identity, sequence သို႔မဟုတ္ hilo အစရွိသည္တို႔ကို အသံုးၿပဳပါတယ္။

.....

<id ...>

<generator class="native"></generator>

</id>

.....

6) identity

id column ကို support လုပ္ရန္ Sybase, My SQL, MS SQL Server, DB2 နဲ႔ HypersonicSQL မ်ားထဲမွာ identity ကို အသံုးၿပဳပါတယ္။ returned id ကေတာ့ short, int or long တို႔ ၿဖစ္ၾကပါတယ္။ unique identifier ကို generate လုပ္ရန္ database ရဲ႕ responsibility ၿဖစ္ပါတယ္။

7) seqhilo

high and low algorithm ကို specified sequence name မွာ အသံုးၿပဳပါတယ္။ type short, int or long ရဲ႕ id ကို return ၿပန္ပါလိမ့္မယ္။

8) uuid

It uses 128-bit UUID algorithm to generate the id. The returned id is of type String, unique within a network (because IP is used). The UUID is represented in hexadecimal digits, 32 in length.

9) guid

It uses GUID generated by database of type string. It works on MS SQL Server and MySQL.

10) select

It uses the primary key returned by the database trigger.

11) foreign

It uses the id of another associated object, mostly used with <one-to-one> association.

12) sequence-identity

It uses a special sequence generation strategy. It is supported in Oracle 10g drivers only.

3. SQL Dialects in Hibernate

SQL statements ရဲ႕ appropriate type ကို hibernate က generate လုပ္ေပးပါတယ္။ hibernate မွာ dialect ဆိုတာက ဘယ္ DB ကို သံုးမလဲဆိုတာကို ဆံုးၿဖတ္ေပးတဲ့အတိုင္းပဲ ၿဖစ္ပါတယ္။ အဲ့တာေၾကာင့္ hibernate က ဘယ္ db ကို မဆိုခ်ိတ္ဆက္မယ္ဆိုရင္ dialects configuration ကို အရင္ခ်ဖို႔ လိုအပ္ပါလိမ့္မယ္။

Syntax of SQL Dialect

<property name="dialect">org.hibernate.dialect.Oracle9Dialect</property>

List of SQL Dialects

|  |  |
| --- | --- |
| RDBMS | Dialect |
| Oracle (any version) | org.hibernate.dialect.OracleDialect |
| MySQL | org.hibernate.dialect.MySQLDialect |
| PostgreSQL | org.hibernate.dialect.PostgreSQLDialect |
| Microsoft SQL Server | org.hibernate.dialect.SQLServerDialect |

**Hibernate Log4j**

1. HB with Log4j 1
2. HB with Log4j 2

1. HB with Log4j 1

Hibernate Logging by Log4j using xml file

Logging ဆိုတာက file တစ္ခုတည္းမွာ log မ်ားကို permanently သြားရိုက္တဲ့ process ကို ေခၚပါတယ္။ hibernate မွာေတာ့ logging process အတြက္ Log4j နဲ႔ Logback frameworks တို႔ကို အသံုးၿပဳႏိုင္ပါတယ္။

log4j ကို နည္းလမ္း ၂ မ်ိဳးနဲ႔ အသံုးၿပဳလို႔ရပါတယ္။

1. xml file
2. properties.file

Levels of Logging

|  |  |
| --- | --- |
| Levels | Description |
| OFF | logging ကို ပိတ္ဖို႔ အသံုးၿပဳပါတယ္။ |
| WARNING | message level တစ္ခုၿဖစ္ၿပီးေတာ့ problem တစ္ခုကို ညႊန္ၿပပါလိမ့္မယ္ |
| SEVERE | message level တစ္ခုၿဖစ္ၿပီးေတာ့ failure တစ္ခုကို ညႊန္ၿပပါလိမ့္မယ္ |
| INFO | informational messages မ်ားအတြက္ အသံုးၿပဳတဲ့ level တစ္ခုၿဖစ္ပါတယ္ |
| CONFIG | static configuration messages မ်ားအတြက္ အသံုးၿပဳတဲ့ level တစ္ခုၿဖစ္ပါတယ္ |

Log4j xml file အသံုးၿပဳၿပီး Hibernate Logging ကိုလုပ္ေဆာင္ရန္ အဆင့္မ်ား

နည္းလမ္း ၂ မ်ိဳးရွိပါတယ္။

1. hibernate နဲ႔ log4j jar files မ်ားကို load ဆြဲတင္
2. src folder ထဲမွာ log4j.xml file ကို create လုပ္။အဲ့မွာ log ရိုက္ေစခ်င္တဲ့ log file ကို လမ္းေၾကာင္းနဲ႔နာမည္နဲ႔ တစ္ခါတည္းေပးထားခဲ့

Log4j xml file အသံုးၿပဳၿပီး Hibernate Logging ကိုလုပ္ေဆာင္တဲ့ example

Hibernate မွာ log ရိုက္မယ္ ဒီ step ၂ ခုကိုေတာ့ လုပ္ကို လုပ္ရမယ္။

Load the required jar files

slf4j.jar နဲ႔ log4j.jar files မ်ားကို hibernate jar files (load လုပ္ၿပီးသား) နဲ႔အတူ load လုပ္ရမွာ ၿဖစ္ပါတယ္။

Create log4j.xml file

log ရိုက္ဖို႔ လမ္းေၾကာင္းနဲ႔ file name ကို xml file ထဲမွာ တစ္ခါတည္း ထည့္ေပးလိုက္ဖို႔ လိုအပ္ပါလိမ့္မယ္။

\*\*ဒီ log file ကို စမ္းမယ္ဆိုရင္ main method ပါတဲ့ hibernate test တစ္ခုခုနဲ႔စမ္းပါ။

log4j.xml

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE log4j:configuration SYSTEM "log4j.dtd">

<log4j:configuration xmlns:log4j="http://jakarta.apache.org/log4j/"debug="false">

<appender name="CONSOLE" class="org.apache.log4j.ConsoleAppender">

<layout class="org.apache.log4j.PatternLayout">

<param name="ConversionPattern" value="[%d{dd/MM/yy hh:mm:ss:sss z}] %5p %c{2}: %m%n" />

</layout>

</appender>

<appender name="ASYNC" class="org.apache.log4j.AsyncAppender">

<appender-ref ref="CONSOLE" />

<appender-ref ref="FILE" />

</appender>

<appender name="FILE" class="org.apache.log4j.RollingFileAppender">

<param name="File" value="C:/Users/Techfunmmr/AppData/javatpointlog.log" />

<param name="MaxBackupIndex" value="100" />

<layout class="org.apache.log4j.PatternLayout">

<param name="ConversionPattern" value="[%d{dd/MM/yy hh:mm:ss:sss z}] %5p %c{2}: %m%n" />

</layout>

</appender>

<category name="org.hibernate">

<priority value="DEBUG" />

</category>

<category name="java.sql">

<priority value="debug" />

</category>

<root>

<priority value="INFO" />

<appender-ref ref="FILE" />

</root>

</log4j:configuration>

2. HB with Log4j 2

Hibernate Logging by Log4j using properties file

Log4j xml file အသံုးၿပဳၿပီး Hibernate Logging ကိုလုပ္ေဆာင္ရန္ အဆင့္မ်ား

1. hibernate နဲ႔ log4j jar files မ်ားကို load ဆြဲတင္
2. src folder ထဲမွာ log4j.properties file ကို create လုပ္။အဲ့မွာ log ရိုက္ေစခ်င္တဲ့ log file ကို လမ္းေၾကာင္းနဲ႔နာမည္နဲ႔ တစ္ခါတည္းေပးထားခဲ့ပါ။

Log4j.properties file အသံုးၿပဳၿပီး Hibernate Logging ကိုလုပ္ေဆာင္တဲ့ example

Load the required jar files

slf4j.jar နဲ႔ log4j.jar files မ်ားကို hibernate jar files (load လုပ္ၿပီးသား) နဲ႔အတူ load လုပ္ရမွာ ၿဖစ္ပါတယ္။

Create log4j.xml file

log ရိုက္ဖို႔ လမ္းေၾကာင္းနဲ႔ file name ကို log4j.properties file ထဲမွာ တစ္ခါတည္း ထည့္ေပးလိုက္ဖို႔ လိုအပ္ပါလိမ့္မယ္။

\*\*ဒီ log file ကို စမ္းမယ္ဆိုရင္ main method ပါတဲ့ hibernate test တစ္ခုခုနဲ႔စမ္းပါ။ ဒါေပမယ့္ xml file ေရာ properties file တစ္ခါတည္း စမ္းလို႔ မရဘူး။ တစ္ခုတည္းထည့္စမ္းလို႔ ရမယ္။ ႏွစ္ခုလံုးထည့္ရင္ေတာ့ xml file ကိုပဲ သံုးသြားလိမ့္မယ္။

log4j.properties

# Direct log messages to a log file

log4j.appender.file=org.apache.log4j.RollingFileAppender

log4j.appender.file.File=C:\\Users\\Techfunmmr\\AppData\\log4jPropertiesTest.log

log4j.appender.file.MaxFileSize=1MB

log4j.appender.file.MaxBackupIndex=1

log4j.appender.file.layout=org.apache.log4j.PatternLayout

log4j.appender.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

# Direct log messages to stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.Target=System.out

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n

# Root logger option

log4j.rootLogger=INFO, file, stdout

# Log everything. Good for troubleshooting

log4j.logger.org.hibernate=INFO

# Log all JDBC parameters

log4j.logger.org.hibernate.type=ALL

**Inheritance Mapping**

1. Inheritance Mapping
2. Table Per Hierarchy
3. TPH using Annotation
4. Table Per Concrete
5. TPC using Annotation
6. Table Per Subclass
7. TPS using Annotation

1. Inheritance Mapping

ကၽြန္ေတာ္တို႔အေနနဲ႔ inheritance hierarchy classes ေတြကို database ထဲက table နဲ႔ map လုပ္လို႔ ရႏိုင္ပါတယ္။ hibernate inheritance ထဲမွာ mapping strategies ၃ မ်ိဳး ရွိပါတယ္။

1. Table Per Hierarchy
2. Table Per Concrete class
3. Table Per Subclass

Table Per Hierarchy

hierarchy တစ္ခုလံုးကို map လုပ္ဖို႔အတြက္ table တစ္လံုးသာလွ်င္ လိုအပ္ပါတယ္။ class မွာ extra column (discriminator column လို႔ေခၚတဲ့) တစ္ခုကို class ကို identify လုပ္ဖို႔ရန္အတြက္ ထပ္ထည့္ဖို႔ေတာ့ လိုပါလိမ့္မယ္။ ဒါေပမယ့္ table ထဲမွာေတာ့ null value ေတြ ၀င္သြားလိမ့္မယ္။

Table Per Concrete class

tables are created as per class ကို ဆိုလိုသည္။ ဒါေပမယ့္ subclass tables ထဲမွာေတာ့ duplicate column ေပါင္းထည့္ရမယ္။

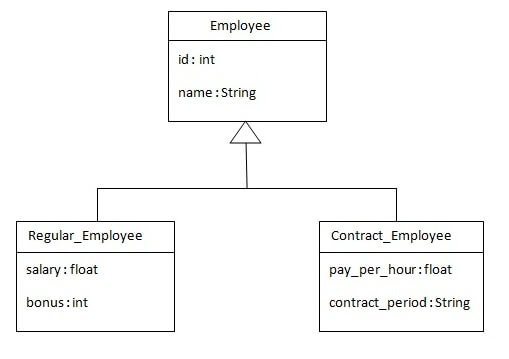
Table Per Subclass

tables are created as per class ကို ဆိုလိုသည္။ ဒါေပမယ့္ foreign key နဲ႔ relate လုပ္ထားပါတယ္။ ဒါေၾကာင့္ ဒီမွာ duplicate columns မ်ား မရွိပါဘူး။

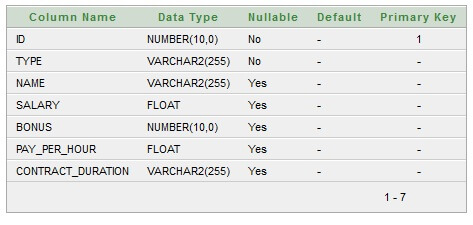
2. Table per Hierarchy (class အမ်ားၾကီး table တစ္လံုး - xml)

Hibernate Table Per Hierarchy using xml file

ကၽြန္ေတာ္တို႔အေနနဲ႔ hierarchy တစ္ခုလံုးကို single table only တစ္ခုတည္းနဲ႔ map လုပ္လို႔ရပါတယ္။ class ကုိ identify လုပ္ဖို႔ရန္ extra column (discriminator column လို႔ေခၚတဲ့) တစ္ခုကို table ထဲမွာေတာ့ create လုပ္ရမယ္။



hierarchy ထဲမွာ class ၃ ခုပါ၀င္ပါတယ္။ Regular\_Employee နဲ႔ Contract\_Employee classes တို႔ေတြအတြက္ Employee class က super class တစ္ခုၿဖစ္ပါတယ္။



Example of Table per class hierarchy

အခု example မွာ class ၃ ခုကို create လုပ္မယ္ ၿပီးရင္ employee.hbm.xml file ထဲမွာ အဲ့ file ၃ file ကို mapping လုပ္ပါမယ္။

1) Create the Persistent classes

Employee1.java

**public** **class** Employee1 {

**private** **int** id;

**private** String name;

getter/setter

Regular\_Employee1.java

**public** **class** Regular\_Employee1 **extends** Employee1 {

**private** **float** salary;

**private** **int** bonus; **getter/setter**

Contract\_Employee1.java

**public** **class** Contract\_Employee1 **extends** Employee1 {

**private** **float** pay\_per\_hour;

**private** String contract\_duration; **getter and setter**

employee1.hbm.xml

<hibernate-mapping>

<class name=*"inheritance\_mapping.table\_per\_hierarchy.Employee1"* table=*"emp121"*

discriminator-value=*"emp"*>

<id name=*"id"*>

<generator class=*"increment"*></generator>

</id>

<discriminator column=*"type"* type=*"string"*></discriminator>

<property name=*"name"*></property>

<subclass name=*"inheritance\_mapping.table\_per\_hierarchy.Regular\_Employee1"*

discriminator-value=*"reg\_emp"*>

<property name=*"salary"*></property>

<property name=*"bonus"*></property>

</subclass>

<subclass name=*"inheritance\_mapping.table\_per\_hierarchy.Contract\_Employee1"*

discriminator-value=*"con\_emp"*>

<property name=*"pay\_per\_hour"*></property>

<property name=*"contract\_duration"*></property>

</subclass>

</class>

</hibernate-mapping>

StoreData1.java

**public** **class** StoreData1 {

**public** **static** **void** main(String[] args) {

StandardServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();

Metadata meta = **new** MetadataSources(ssr).getMetadataBuilder().build();

SessionFactory factory = meta.getSessionFactoryBuilder().build();

Session session = factory.openSession();

Transaction t = session.beginTransaction();

Employee1 e1 = **new** Employee1();

e1.setName("Gaurav Chawla");

Regular\_Employee1 e2 = **new** Regular\_Employee1();

e2.setName("Vivek Kumar");

e2.setSalary(50000);

e2.setBonus(5);

Contract\_Employee1 e3 = **new** Contract\_Employee1();

e3.setName("Arjun Kumar");

e3.setPay\_per\_hour(1000);

e3.setContract\_duration("15 hours");

session.persist(e1);

session.persist(e2);

session.persist(e3);

t.commit();

session.close();

System.***out***.println("success");}}

3. TPH using Annotation (class အမ်ားၾကီး table တစ္လံုး - annotation)

Hibernate Table Per Hierarchy using annotation

Employee2.java

@Entity

@Table(name = "employee101")

@Inheritance(strategy = InheritanceType.***SINGLE\_TABLE***)

@DiscriminatorColumn(name = "type", discriminatorType = DiscriminatorType.***STRING***)

@DiscriminatorValue(value = "employee2")

**public** **class** Employee2 {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

@Column(name = "id")

**private** **int** id;

@Column(name = "name")

**private** String name; **//getter and setter**

Regular\_Employee2.java

@Entity

@DiscriminatorValue("regularemployee2")

**public** **class** Regular\_Employee2 **extends** Employee2 {

@Column(name = "salary")

**private** **float** salary;

@Column(name = "bonus")

**private** **int** bonus; **//getter and setter**

Contract\_Employee2.java

@Entity

@DiscriminatorValue("contractemployee2")

**public** **class** Contract\_Employee2 **extends** Employee2 {

@Column(name = "pay\_per\_hour")

**private** **float** pay\_per\_hour;

@Column(name = "contract\_duration")

**private** String contract\_duration; **//getter and setter**

hibernate.cgf.xml

<!-- inheritance\_mapping.tph\_using\_annotation -->

<mapping class=*"inheritance\_mapping.tph\_using\_annotation.Employee2"*/>

<mapping class=*"inheritance\_mapping.tph\_using\_annotation.Regular\_Employee2"*/>

<mapping class=*"inheritance\_mapping.tph\_using\_annotation.Contract\_Employee2"*/>

StoreTest.java

**public** **class** StoreTest2 {

**public** **static** **void** main(String args[]) {

StandardServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();

Metadata meta = **new** MetadataSources(ssr).getMetadataBuilder().build();

SessionFactory factory = meta.getSessionFactoryBuilder().build();

Session session = factory.openSession();

Transaction t = session.beginTransaction();

Employee2 e1 = **new** Employee2();

e1.setName("Gaurav Chawla");

Regular\_Employee2 e2 = **new** Regular\_Employee2();

e2.setName("Vivek Kumar");

e2.setSalary(50000);

e2.setBonus(5);

Contract\_Employee2 e3 = **new** Contract\_Employee2();

e3.setName("Arjun Kumar");

e3.setPay\_per\_hour(1000);

e3.setContract\_duration("15 hours");

session.persist(e1);

session.persist(e2);

session.persist(e3);

t.commit();

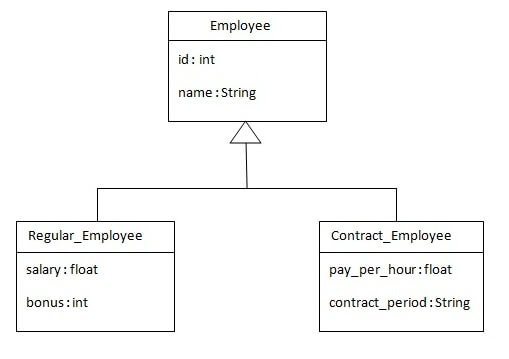
session.close();

System.***out***.println("success");}}

4. Table Per Concrete (class တစ္ခု table တစ္လံုး parent table field မ်ား child table ထဲေရာက္ - xml)

database ထဲမွာ တစ္ခုနဲ႔ တစ္ခု relation မရွိတဲ့ table ၃ လံုး ရွိေနလိမ့္မယ္။ table per concrete class strategy ကို နည္းလမ္း ၂ မ်ိဳးနဲ႔ table နဲ႔ map လုပ္လို႔ရႏိုင္တယ္။

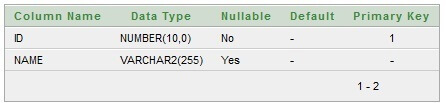
* By union-subclass element
* By self creating the table for each class



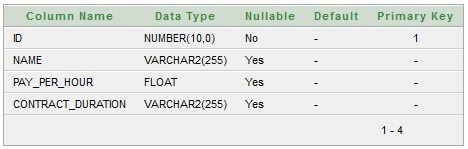
employee3.hbm.xml

database ထဲမွာ table ၃ လံုးရွိၿပီးေတာ့ အဲ့ table တစ္လံုးခ်င္းစီက class တစ္ခုခ်င္းစီကို ကိုယ္စားၿပဳပါတယ္။ class ရဲ႕ union-subclass subelement က sub class ကို သတ္မွတ္ေပးပါတယ္။ parent class ရဲ႕ column ေတြၿဖစ္ၾကတဲ့ id နဲ႔ name ကို sub table မ်ားထဲမွာ သြားထည့္ေပးရမယ္။

Employee3 class ရဲ႕ table structure



Contract\_Employee3 class ရဲ႕ table structure



Regular\_Employee3 class ရဲ႕ table structure



Example of Table per concrete class

ဒီ example မွာဆိုရင္ class ၃ ခုကို create လုပ္မယ္။ employee3.hbm.xml file ထဲမွာ အဲ့ class ေတြကို mapping လုပ္ပါမယ္။

1) Create the Persistent classes

Employee3.java

**public** **class** Employee3 {

**private** **int** id;

**private** String name;

Regular\_Employee3.java

**public** **class** Regular\_Employee3 **extends** Employee3{

**private** **float** salary;

**private** **int** bonus;

Contract\_Employee3.java

public class Contract\_Employee3 extends Employee3 {

private float pay\_per\_hour;

private String contract\_duration;

2) Create the mapping file for Persistent class

employee3.hbm.xml

<hibernate-mapping>

<class name=*"inheritance\_mapping.table\_per\_con\_concrete.Employee3"* table=*"emp122"*>

<id name=*"id"*>

<generator class=*"increment"*></generator>

</id>

<property name=*"name"*></property>

<union-subclass

name=*"inheritance\_mapping.table\_per\_con\_concrete.Regular\_Employee3"* table=*"regemp122"*>

<property name=*"salary"*></property>

<property name=*"bonus"*></property>

</union-subclass>

<union-subclass

name=*"inheritance\_mapping.table\_per\_con\_concrete.Contract\_Employee3"* table=*"contemp122"*>

<property name=*"pay\_per\_hour"*></property>

<property name=*"contract\_duration"*></property>

</union-subclass>

</class>

</hibernate-mapping>

hibernate.cfg.xml

<!-- inheritance\_mapping.table\_per\_con\_concrete -->

<mapping resource=*"inheritance\_mapping/table\_per\_con\_concrete/employee3.hbm.xml"* />

StoreData3.java

**public** **class** StoreData3 {

**public** **static** **void** main(String[] args) {

StandardServiceRegistry ssr = **new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();

Metadata meta = **new** MetadataSources(ssr).getMetadataBuilder().build();

SessionFactory factory = meta.getSessionFactoryBuilder().build();

Session session = factory.openSession();

Transaction t = session.beginTransaction();

Employee3 e1 = **new** Employee3();

e1.setName("Gaurav Chawla");

Regular\_Employee3 e2 = **new** Regular\_Employee3();

e2.setName("Vivek Kumar");

e2.setSalary(50000);

e2.setBonus(5);

Contract\_Employee3 e3 = **new** Contract\_Employee3();

e3.setName("Arjun Kumar");

e3.setPay\_per\_hour(1000);

e3.setContract\_duration("15 hours");

session.persist(e1);

session.persist(e2);

session.persist(e3);

t.commit();

session.close();

System.***out***.println("success");}}

5. TPC Using Annotation (class တစ္ခု table တစ္လံုး parent table field မ်ား child table ထဲေရာက္ - annotation)

Table Per Concrete class using Annotation

Table Per Concrete class မွာဆိုရင္ class တစ္ခုကို table တစ္လံုး create လုပ္ပါမယ္။ အဲ့တာေၾကာင့္ table ထဲမွာ nullable values မ်ား မရွိပါဘူး။ ဒီမွာ မေကာင္းတဲ့အခ်က္ကေတာ့ sub tables မ်ားထဲတြင္ duplicate columns မ်ား create လုပ္လို႔ရေနၿခင္းပဲ ၿဖစ္ပါတယ္။

parent class ထဲမွာ @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS) ဆိုတဲ့ annotation ကို သံုးဖို႔ လိုအပ္ပါတယ္။

sub class ထဲမွာေတာ့ @AttributeOverrides ဆိုတဲ့ annotation ကို သံုးဖို႔လိုအပ္ပါတယ္။

* table per concrete class strategy ကို သံုးမယ္ဆိုရင္ @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS) ဆိုတဲ့ annotation ကို သံုးကို သံုးေပးရမယ္။ သူ႕ကိုေတာ့ parent class ထဲမွာပဲ သံုးရမွာ ၿဖစ္ပါတယ္။
* sub class ေတြထဲမွာ parent class attributes မ်ားကို overridden လုပ္မယ္ဆိုရင္ @AttributeOverride ဆိုတဲ့ annotation ကို သံုးေပးရမယ္။ အဲ့လိုဆိုရင္ parent class ေတြရဲ႕ table column ေတြက subclass table ေတြထဲမွာ သြားေပါင္းလိမ့္မယ္။

Example of Table per concrete class

Employee4.java

@Entity

@Table(name = "employee102")

@Inheritance(strategy = InheritanceType.***TABLE\_PER\_CLASS***)

**public** **class** Employee4 {

@Id

@GeneratedValue(strategy = GenerationType.***AUTO***)

@Column(name = "id")

**private** **int** id;

@Column(name = "name")

**private** String name;

Regular\_Employee4 .java

@Entity

@Table(name = "regularemployee102")

@AttributeOverrides({ @AttributeOverride(name = "id", column = @Column(name = "id")),

@AttributeOverride(name = "name", column = @Column(name = "name")) })

**public** **class** Regular\_Employee4 **extends** Employee4 {

@Column(name = "salary")

**private** **float** salary;

@Column(name = "bonus")

**private** **int** bonus;

Contract\_Employee4 .java

@Entity

@Table(name = "contractemployee102")

@AttributeOverrides({ @AttributeOverride(name = "id", column = @Column(name = "id")),

@AttributeOverride(name = "name", column = @Column(name = "name")) })

**public** **class** Contract\_Employee4 **extends** Employee4 {

@Column(name = "pay\_per\_hour")

**private** **float** pay\_per\_hour;

@Column(name = "contract\_duration")

**private** String contract\_duration;

hibernate.cfg.xml

<!-- inheritance\_mapping.tpc\_using\_annotation -->

<mapping class=*"inheritance\_mapping.tpc\_using\_annotation.Employee4"*/>

<mapping class=*"inheritance\_mapping.tpc\_using\_annotation.Contract\_Employee4"*/>

<mapping class=*"inheritance\_mapping.tpc\_using\_annotation.Regular\_Employee4"*/>

StoreData4 .java

**public** **class** StoreData4 {

**public** **static** **void** main(String[] args) {

StandardServiceRegistry ssr=**new** StandardServiceRegistryBuilder().configure("hibernate.cfg.xml").build();

Metadata meta=**new** MetadataSources(ssr).getMetadataBuilder().build();

SessionFactory factory=meta.getSessionFactoryBuilder().build();

Session session=factory.openSession();

Transaction t=session.beginTransaction();

Employee4 e1=**new** Employee4();

e1.setName("Gaurav Chawla");

Regular\_Employee4 e2=**new** Regular\_Employee4();

e2.setName("Vivek Kumar");

e2.setSalary(50000);

e2.setBonus(5);

Contract\_Employee4 e3=**new** Contract\_Employee4();

e3.setName("Arjun Kumar");

e3.setPay\_per\_hour(1000);

e3.setContract\_duration("15 hours");

session.persist(e1);

session.persist(e2);

session.persist(e3);

t.commit();

session.close();

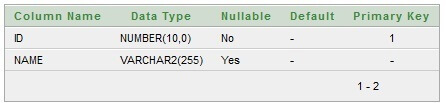
System.***out***.println("success"); } }

6. Table Per Subclass (class တစ္ခု table တစ္လံုး parent table ရဲ႕ id က child table ထဲမွာ fk ၿဖစ္ - xml)

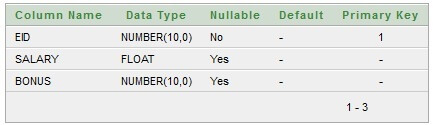
table per subclass မွာဆိုရင္ primary key foreign key relationship ပံုစံနဲ႔သြားမွာ ၿဖစ္ပါတယ္။ အဲ့လို join မယ္ဆိုရင္ <joined-subclass> element ကို သံုးရမွာ ၿဖစ္ပါတယ္။

* joined-subclass ကို subclass အၿဖစ္ သတ္မွတ္ဖို႔ အသံုးၿပဳပါတယ္။
* key sub-element ကိုေတာ့ sub ထဲမွာ parent ရဲ႕ id ကို foreign key အၿဖစ္ သတ္မွတ္ဖို႔ရန္ အသံုးၿပဳပါတယ္။

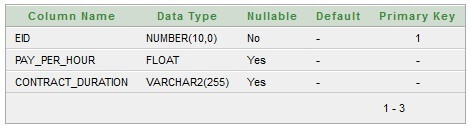
Employee5 class



Regular\_Employee5 class



Contract\_Employee5 class



Example of Table per subclass class

**Hibernate Mapping**

1. Collection Mapping
2. Mapping List
3. Mapping Bag
4. Mapping Set
5. Mapping Map
6. One To Many XML
7. One To Many Annotation
8. Many To Many XML
9. Many To Many Annotation
10. One To One XML
11. One To One Annotation
12. Many To One XML
13. Many To One Annotation
14. Bidirectional
15. Lazy Collection
16. Component Mapping

1. Collection Mapping

ကၽြန္ေတာ္တို႔က hibernate ထဲမွာ persistent class ရဲ႕ collection elements မ်ားကို map လုပ္ေဆာင္ႏိုင္ပါတယ္။ ဒါေပမယ့္ ေအာက္က collection type တစ္ခုခုကိုေတာ့ Persistent class ထဲမွာ declare လုပ္ေပးဖို႔ေတာ့ လိုအပ္ပါလိမ့္မယ္။

1. java.util.List
2. java.util.Set
3. java.util.SortedSet
4. java.util.Map
5. java.util.SortedMap
6. java.util.Collection
7. or write the implementation of org.hibernate.usertype.UserCollectionType

collection element အတြက္ persistent class ကိုေအာက္ေဖာ္ၿပပါအတိုင္း ေရးႏိုင္ပါတယ္။

**import** java.util.List;

**public** **class** Question {

**private** **int** id;

**private** String qname;

**private** List<String> answers;//List can be of any type

  //getters and setters  }

mapping file ထဲက mapping collection

collection ကို map လုပ္ဖို႔ရန္ <class> elements ထဲမွာ subelements မ်ားစြာ ရွိပါတယ္။ သူတို႔ေတြကေတာ့ <list>, <bag>, <set> နဲ႔ <map> တို႔ ၿဖစ္ၾကပါတယ္။

<class name="com.javatpoint.Question" table="q100">

<id name="id">

<generator class="increment"></generator>

</id>

<property name="qname"></property>

<list name="answers" table="ans100">

<key column="qid"></key>

<index column="type"></index>

<element column="answer" type="string"></element>

</list>

</class>

list ထဲမွာ sub elements ၃ ခုကို အသံုးၿပဳႏိုင္ပါတယ္။

1. <key> ဆိုတဲ့ element ကို ေအာက္မွာ ၿပထားတဲ့အတိုင္းေၿပာရရင္ Question class identifier ကို အေၿခခံၿပီး အဲ့ table ထဲမွာ foreign key သတ္မွတ္ရန္ အသံုးၿပဳပါတယ္။
2. <index> ဆိုတဲ့ element ကို type ကို identify လုပ္ရန္အသံုးၿပဳပါတယ္။ List နဲ႔ Map ဟာ indexed collection မ်ားၿဖစ္ၾကပါတယ္။
3. <element> collection ရဲ႕ element ကို define လုပ္ဖို႔ရန္ အသံုးၿပဳပါတယ္။

collection က string objects မ်ားကို store လုပ္ခဲ့မယ္ဆိုရင္ အဲ့တာက collection ရဲ႕ mapping ၿဖစ္ပါတယ္။ ဒါေပမယ့္ အကယ္လို႔မ်ား collection က entity reference (another class objects) မ်ားကို store လုပ္ခဲ့မယ္ဆိုရင္ ကၽြန္ေတာ္တို႔အေနနဲ႔ <one-to-many> သို႔မဟုတ္ <many-to-many> element ဆိုတဲ့ element မ်ားကို သတ္မွတ္ဖို႔ လိုအပ္ပါလိမ့္မယ္။

Question.java

import java.util.List;

public class Question {

private int id;

private String qname;

private List<Answer> answers;//Here, List stores the objects of Answer class

//getters and setters }

Answer.java

import java.util.List;

public class Answer {

private int id;

private String answer;

private String posterName;

//getters and setters

}

mapping file ကေတာ့ ဒီလို ၿဖစ္လိမ့္မယ္။

<class name="com.javatpoint.Question" table="q100">

<id name="id">

<generator class="increment"></generator>

</id>

<property name="qname"></property>

<list name="answers" >

<key column="qid"></key>

<index column="type"></index>

<one-to-many class="com.javatpoint.Answer" />

</list>

</class>

ဒီမွာ list ကို one to many relationship နဲ႔ခ်ိတ္ထားတယ္ ဆိုလိုတာက question တစ္ခုမွာ answer အမ်ားၾကီး ရွိရမယ္။

Understanding key element

key element ကို original identity ကို အေၿခခံထားတဲ့ joint table ထဲမွာ foreign key define လုပ္ဖို႔ရန္ အသံုးၿပဳပါတယ္။ foreign key က default အားၿဖင့္ nullable ၿဖစ္ပါတယ္။ ဆိုေတာ့ foreign key ကို non nullable ၿဖစ္ခ်င္တယ္ဆိုရင္ေတာ့ ေအာက္ကလိုမ်ိဳးသတ္မွတ္ေပးရမယ္။

<key column="qid" not-null="true" ></key>

key element ရဲ႕ attributes ေတြဟာ column, on-delete, property-ref, not-null, update နဲ႔ unique ၿဖစ္ၾကပါတယ္။

<key

column="columnname"

on-delete="noaction|cascade"

not-null="true|false"

property-ref="propertyName"

update="true|false"

unique="true|false"

/>

Indexed collections

indexed collections ကို form ၂ မ်ိဳးဆိုၿပီးေတာ့ category ခြဲထားပါတယ္။

1. indexed
2. non-indexed

List နဲ႔ Map collection က indexed မ်ားၿဖစ္ၾကၿပီးေတာ့ set နဲ႔ bag collections မ်ားကေတာ့ non-indexed မ်ားၿဖစ္ၾကပါတယ္။ indexed collection ဆိုတာက List နဲ႔ Map က additional element <index> တစ္ခုေပါင္းထည့္ဖို႔ လိုအပ္တယ္လို႔ ဆိုလိုပါတယ္။

Collection Elements

ollection elements ေတြ value သို႔မဟုတ္ entity reference (another class object) ရွိပါတယ္။ ကၽြန္ေတာ္တို႔အေနနဲ႔ element ၄ ခုကို အသံုးၿပဳႏိုင္ပါတယ္။

1. element
2. component-element
3. one-to-many, သို႔မဟုတ္
4. many-to-many

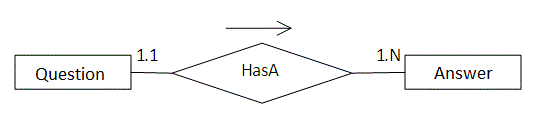
element နဲ႔ component-element ေတြကို သာမန္ value ေတြအတြက္သာ အသံုးၿပဳပါတယ္ ဥပမာ string, int etc။ one-to-many နဲ႔ many-to-many တို႔ဆိုရင္ေတာ့ map လုပ္ဖို႔ရန္ entity reference မ်ားကို အသံုးၿပဳပါတယ္။

2. Mapping List

Mapping List in Collection Mapping (using xml file)

ကၽြန္ေတာ္တို႔ရဲ႕ persistent object မွာ list object ရွိခဲ့တယ္ဆိုရင္ <list> element ကိုသံုးၿပီးေတာ့ mapping file ထဲမွာၿဖစ္ၿဖစ္ annotation ကိုသံုးၿပီးေတာ့ၿဖစ္ၿဖစ္ map လုပ္လိုက္လို႔ရပါတယ္။

question တစ္ခုမွာ answer ေတြအမ်ားၾကီး ရွိတဲ့သေဘာ။



Example of mapping list in collection mapping

1) Create the Persistent class