**Hibernate:**

DAO(Data Access Operation): It helps in connecting with the database.

Hibernate is used for Enterprise Application.

It is developed after finding difficulties with EJB.

If you go with the simple java code(Servlets, JDBC) the business will go to loss as the security provided by them is not up to the mark.

For example if you are transferring money from one account to the other account and one of the system fails. If this happens then whole process should have to be roll back and if you are using simple java then roll back will not happen as it will require transaction service. So business applications require some great capabilities.

To add some great capabilities we have to use service which can be bought from application servers(Jboss, GlassFish, WebLogic etc). simple java will not allow to use services therefore we use EJB’s to buy services from application servers.

If we use jdbc, we have to use statements, preparedstatements , callableStatements. Suppose you are using oracle database then you have to write the queries that are to be supported by the Oracle database. In future you want to shift application to another database then those queries would not be compatible with it thus making it database dependent.

Suppose in a company employees swipe in and swipe out. As jdbc doesnot support caching it will retrieve the data again and again. To improve the performance we can use cache service which will give the capablility to store the data in temporary cache. EJB provides cache support which uses queries that are independent to the databases.

There are many problems with EJB.

Suppose if you want to register a new employee with only few information that does not requires security constraints with using EJB’s entity beans then the controller cannot directly insert that into the database as it have to contact session bean and then entity bean and only after that it can register. Therefore hibernate is introduced.

So the main disadvantage with EJB is that the entity bean will require the session bean and they both will require application server. Hence for small scale application EJB’s are not recommended as they will cost very much because you have to buy many things like Application servers etc and maintenance cost will also be higher .

In hibernate you don’t require any application server. Simple JDK and JRE will work.

Hence if you have 1000 classes that require services and 1000classes that do not require any kind of services then controller can directly contact to hibernate without making contact with session bean. But still there is a dependency as for providing the services application server is to be bought and a separate session bean is to be used. We can reduce this dependency by using Spring. As with spring we don’t have to buy separate application server as spring itself provides that capability

**Features of hibernate:**

1. Auto DDL
2. HQL support(
3. Cache support
4. Primary key generator support
5. Data validation support(by using annotations)
6. Exception free (We don’t need to handle any compile time exception)
7. ORM support(object relation mapping)
8. Inbuilt connection pool.

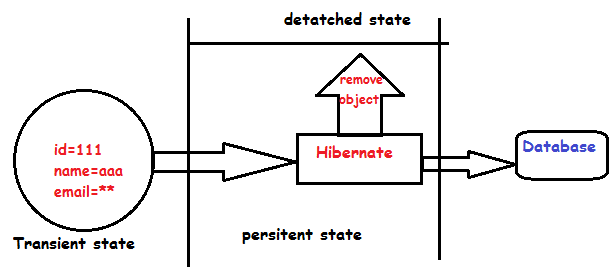
**From hibernate 3 onwards:**

1. OGM(object graph mapping) support(non sql machine)
2. searching

**NOTE:In hibernate you can maintain object to object relationship rather than table to table relationship.**

**Object states in hibernate:**

1. transient
2. persistent
3. detached



We require the following in hibernate:

1. POJO class
2. Mapping file
3. Configuration
4. Test class

POJO class:

* In pojo class you need to have all the properties that needs to be stored in the database.
* Default constructor is mandatory in POJO class.
* Setter and getter methods must be there.
* Setter and getter access should be public.

Public class Student

{

private int id;

private String name;

private String email;

private String marks;

//mandatory default constructor i.e, provided by the class

//public getter and setter

}

Now we have to create mapping file as all it contains all the information as to where the data must be stored. To map the above bean class to the table we need mapping file.

**Student.hbm.xml :::::**

<hibernate-mapping>

<class name=”Student” table=”Student008” schema=”System”>

<id name=”id” column=”sid”/>//for primary key

<property name=”name” column=”name”/>

<property name=”email” column=”email”/>

<property name=”marks column=”marks”/>

Or

<Composite-id>

<id name=”id” column=”sid”/>

</Composite-id>

</class>

</hibernate-mapping>

1. **Configuration File:**

**hibernate.cfg.xml**

< hibernate-configuration>

< /hibernate-configuration>

Create a hibernate 1st running application:

1. ***CREATE POJO student.java***

**package** beans;

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

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

**private** String name;

**private** String email;

**private** String marks;

**//GETTER AND SETTERS**

1. ***CREATE ITS MAPPING FILE student.hbm.xml***

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">

**//The above will be found in jar named hibernate-core => org.hibernate => hibernate-mapping.dtd**

<hibernate-mapping>

<class name=*"beans.Student"* table=*"student007"* schema=*"student"*>

<id name=*"id"* column=*"sid"* type=*"int"*></id>

<property name=*"name"* column=*"sname"* type=*"string"*></property>

<property name=*"email"* column=*"semail"* type=*"string"*></property>

<property name=*"marks"* column=*"smarks"* type=*"string"*></property>

</class>

</hibernate-mapping>

1. ***CREATE ITS CONFIGURATION FILE hibernate.cfg.xml***

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

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

<property name=*"connection.url"*>jdbc:mysql://localhost:3306/student</property>

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

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

<property name=*"connection.pool\_size"*>10</property>

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

<mapping resource=*"resources/student.hbm.xml"*/>

</session-factory>

</hibernate-configuration>

1. ***Create POJO CLASS client.java***

**package** beans;

**import** org.hibernate.Session;

**import** org.hibernate.SessionFactory;

**import** org.hibernate.cfg.Configuration;

**public** **class** Client {

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

Student st=**new** Student();

st.setId(111);

st.setName("Tanuj");

st.setEmail("tanuj.tiwari99@gmail.com");

st.setMarks("78");

//Student object state is transient

Configuration cfg=**new** Configuration();

cfg.configure("resources/hibernate.cfg.xml");

SessionFactory sf=cfg.~~buildSessionFactory~~();

Session s=sf.openSession();

s.save(st);

//Student object state is Persistent

s.beginTransaction().commit();

//student object will move to database

s.evict(st);

//student object will be deleted from session(Persistent)

//now gc can collect student object

}

}

1. **Configuration cfg=new Configuration();**

**The configuration class above is hibernate container class.**

1. **cfg.configure(“resources/hibernate.cfg.xml”);**

**This will load the xml file of configuration in to the container.**

1. **SessionFactory sf=cfg.buildSessionFactory();**

**This will drop/create/update the tables.**

**NOTE: The above program will first drop the tables if already present and then it will create the new tables. In case if you do not want to delete the existing table you can use UPDATE instead of CREATE.**

**IMPORTANT:**

1. **Suppose you have dropped one table and you want to again create it. If you will run the same program it will first drop the tables and then create all the tables. So instead of using CREATE again you got to use UPDATE.**
2. **Suppose you want to add another column the student table then you don’t have to create that whole table again because you can use the UPDATE provided you should define the column in the POJO student.java and map it in student.hbm.xml. but it has one limitation that is if the table contains the data then hibernate will not be able to assign the new column with its default value NULL. So altering is not possible if the table already contains the data.**

If we want to use AUTO DDL operation in which tables and sql queries will be automatically generated we can use ::::

1. **Create**
2. **Update**
3. **Validate**
4. **Create-update**

<property name=*"hbm2ddl.auto"*>create</property> in hibernate.cfg.xml

Here create will create the table which has been mapped in student.hbm.xml which provides the mapping of POJO class Student.java.

If we want to see the query also then we can add

<property name=*"show\_sql"*>true</property>

This above will show the queries in the console.

**NOTE: the tables will contain the field with the same name that has been provided in the POJO class. In case you want to have different field names then you can use the column attribute to give the different name .**

***CURD operations:::***

There are 3 methods to inset the data in the table.

1. save(): It will return the primary key
2. persist(): It returns void
3. saveOrUpdate(): It returns void

benefit of saveOrUpdate operation is that it uses select query using primary key first and checks if there is already all the records present. If all the records are present it will not do anything

if any record is different then it will do update .

if the primary key is changed then it will save.

For example:

**Student st=new Student();**

**st.setId(1);**

**st.setName(“Tanuj”);**

**Configuration cfg=new Configuration();**

**Cfg.configure(“hibernate.cfg.xml”);**

**SessionFactory sf=cfg.buildSessionFactory();**

**Session s=sf.openSession();**

**s.saveOrUpdate(st);**

In the above if both the record will be same then it will not do anything.

Suppose if name is changed then it will do update.

If primary key is changed then it will save.

**update()::::::**

1. **it is used to update the values in the table.**
2. **It can only update non primary key values.**
3. **It is not possible to update one column. If you forget to write the total values then it will insert null value.**

**delete(): It is used for deletion**

**For select operations::::**

**get(Student.class,primary key);**

**load(Student.class,primary key);**

Configuration cfg=**new** Configuration();

cfg.configure("resources/hibernate.cfg.xml");

SessionFactory sf=cfg.~~buildSessionFactory~~();

Session s=sf.openSession();

Object o=s.get(Student.**class**,12);//It is a select query

**OR**

Object o=s.load(Student.**class**,12);//It is a select query

Student stu=(Student)o;

String name=stu.getName();

String roll=stu.getRoll();

System.*out*.println(name+":::::"+roll);

**Primary key auto generator:::::**

**There are various properties that are used to generate primary key automatically.**

1. **Assigned**
2. **Increment**
3. **Identity**
4. **Sequence**
5. **Hilo**
6. **Native**
7. **Foreign**
8. **customGenators**
9. **In case of assigned user is responsible to provide the primary key. These are used because in case of registering user can enter the same id multiple times.**
10. **In this case of Increment you don’t have to provide the primary key, when you store the data in the database automatically it will start with 1 as a primary key and increment by one each time when you store it in the database.**
11. **This is used in Msql database**
12. **In the Sequence you need to create a sequence in the database::**

**Sequence is used in oracle database.**

**CREATE SEQUENCE jntu\_sequence**

**START WITH 1220**

**INCREMENT BY 1**

**NOCYCLE**

**NOCACHE;**

**IN student.hbm.xml::::**

<id name=*"id"*>

<generator class=*"sequence"*>

<param name=”sequence”>*jntu\_sequence*</param>

</generator>

</id>

**HQL:**

We can use CRUD(Insert,update,delete,select) operations using hibernate query language.

In update() method and merge() method we cannot update primary key.

And also it is not possible to update one single column.

But with the HQL it is possible to update primary key as well as one single column.

Sql queries are database dependent while hql queries are databse independent.

Hql are object queries while sql are table queries

For example in HQL

**Select name from Student, here Student is a class.**

**INSERT:**

Limitation of hql insert query is that we cannot insert used defined values in the table.

HQL queries can transfer the data from one table to the another table.

**Configuration cfg=new Configuration();**

**Cfg.configure(“resources/hibernate.cfg.xml”);**

**SessionFactory sf=cfg.buidSessionFactory();**

**Session s=sf.openSession();**

**Transaction t=s.beginTransaction();**

**String query = INSERT INTO newStudent(id, name, email, roll) SELECT s.id, s.name, s.email, s.roll FROM oldStudent s;**

**String query = UPDATE STUDENT SET id=444 where id=555; //for updating primary key**

**String query = DELETE STUDENT where id=555; //for updating primary key**

**Query q=s.createQuery(query); //returns query**

**Int i=q.executeUpdate(); //returns the number of rows which are inserted**

**t.commit();**

**s.close();**

**sf.close();**

In the above all the data from the oldStudent table is dumped into newStudent table.

**Hibernate JavaBrains**

* An Orm tool
* Used in data layer of applications
* Implements JPA
* JPA stands for java persistant API. It is the set of standard that have been prescribed for any persitance implementation that need to me met in order to get certified as JPA specification.

The problem before was that we have to save java objects into the relational table. These two are two different things, so in order to do that we had to write hell of a boiler plate code as all those java objects are to be converted so that they can be persisted into the relational table.

Entity class:

@Entity

**public** **class** UserDetails {

@Id

**private** **int** userId;

**private** String userName;

**// Getter and Setters**

}

Test class

**public** **class** HibernateTest {

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

UserDetails userDetails = **new** UserDetails();

userDetails.setUserId(2);

userDetails.setUserName("Tanuj");

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

Session session = sessionFactory.openSession();

session.beginTransaction();

session.save(userDetails);

session.getTransaction().commit();

}}

Hibernate.cfg.xml:

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

<session-factory>

<!-- Database connection settings -->

<property name=*"connection.driver\_class"*>org.postgresql.Driver</property>

<property name=*"connection.url"*>jdbc:postgresql://localhost:5432/hibernatedb

</property>

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

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

<!-- JDBC connection pool (use the built-in) -->

<property name=*"connection.pool\_size"*>1</property>

<!-- SQL dialect -->

<property name=*"dialect"*>org.hibernate.dialect.PostgreSQLDialect

</property>

<!-- Enable Hibernate's automatic session context management -->

<property name=*"current\_session\_context\_class"*>thread</property>

<!-- Disable the second-level cache -->

<property name=*"cache.provider\_class"*>org.hibernate.cache.internal.NoCacheProvider

</property>

<!-- Echo all executed SQL to stdout -->

<property name=*"show\_sql"*>true</property>

<!-- Drop and re-create the database schema on startup -->

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

<mapping class=*"com.hibernate.dto.UserDetails"* />

</session-factory>

</hibernate-configuration>

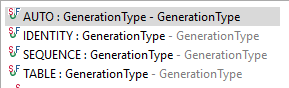
Entity class:

@Entity

**public** **class** UserDetails {

@Id @GeneratedValue **// by default it is auto**

**private** **int** userId;



@Transient **// hibernate won’t consider this field**

**private** **String** userName;

@Temporal(TemporalType.***DATE***) **// date will we saved instead of whole timestamp**

**private** **String** userName;

@Lob **// large object**

@Column(name = "DESCRIPTION")

**private** String description;

**// Getter and Setters**}

**Embedding Objects**

If we want to embed an object that also contains several fields then we have to use @Embeddable in the class which we want to include in the Entity class and in the entity class we have to use @Embedded annotation.

@Embeddable

**public** **class** Address {

**private** String street;

**private** String city;

**private** String state;

**private** String pincode;

**// Getter and setters**

}

@Entity @GeneratedValue

**public** **class** UserDetails {

@Id

**private** **int** userId;

@Embedded **// Now, the Address class is embedded in the entity class**

**private** Address address;}

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

UserDetails userDetails = **new** UserDetails();

Address address = **new** Address("many", "bang", "kar", "560045");

userDetails.setUserName("Tanuj");

userDetails.setAddress(address);

SessionFactory sf = **new** Configuration().configure().buildSessionFactory();

Session session = sf.openSession();

session.beginTransaction();

session.save(userDetails);

session.getTransaction().commit();

session.close();

}

@Table(name = "USER\_DETAILS")

**public** **class** UserDetails {

//@EmbeddedId **// if you want an object to be primary key**

**Private LoginId loginId;**

@Id

@GeneratedValue

@Column(name = "USER\_ID")

**private** **int** userId;

@Column(name = "USER\_NAME")

**private** String userName;

@Embedded

@AttributeOverrides({

@AttributeOverride(name = "street", column = @Column(name = "HOME\_STREET")),

@AttributeOverride(name = "city", column = @Column(name = "HOME\_CITY")),

@AttributeOverride(name = "state", column = @Column(name = "HOME\_state")),

@AttributeOverride(name = "pincode", column = @Column(name = "HOME\_PINCODE"))})

**private** Address homeAddress;

@Embedded

@AttributeOverrides({

@AttributeOverride(name = "street", column = @Column(name = "OFFICE\_STREET")),

@AttributeOverride(name = "city", column = @Column(name = "OFFICE\_CITY")),

@AttributeOverride(name = "state", column = @Column(name = "OFFICE\_state")),

@AttributeOverride(name = "pincode", column = @Column(name = "OFFICEE\_PINCODE"))})

**private** Address officeAddress;

}

Collections

Hibernate looks how many Collection elements are there and creates separate tables for each of them.

Entity class

@Entity

@Table(name = "USER\_DETAILS")

**public** **class** UserDetails {

@Id

**private** **int** userId;

@ElementCollection **--------------------------------(1)**

@JoinTable(name="USER\_ADDRESS", joinColumns=@JoinColumn(name="USER\_ID"))

**// Now the name of the table will be USER\_ADDRESS instead of USERDETAILS\_list**

**// And the name of primary key column will be USER\_ID instead of userdetails\_user\_id**

**private** List<Address> list = **new** ArrayList<Address>();

}

**#NOTE: If we want to create primary key in that collection table USER\_ADDRESS then we have to use hibernate provided annotaions:**

@ElementCollection**--------------------------------(2)**

@JoinTable(name="USER\_ADDRESS", joinColumns=@JoinColumn(name="USER\_ID"))

@GenericGenerator(name = "sequence-gen", strategy = "sequence")

@CollectionId(columns = { @Column(name="ADDRESS\_ID") }, generator = "sequence-gen", type = @Type(type="long"))

**private** Collection<Address> list = **new** ArrayList<Address>();

HibernateTest class

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

UserDetails userDetails = **new** UserDetails();

Address homeAddress = **new** Address("many", "bang", "kar", "560045");

Address officeAddress = **new** Address("off", "off", "off", "off");

userDetails.setUserName("Tanuj");

userDetails.getList().add(homeAddress);

userDetails.getList().add(officeAddress);

SessionFactory sf = **new** Configuration().configure().buildSessionFactory();

Session session = sf.openSession();

session.beginTransaction();

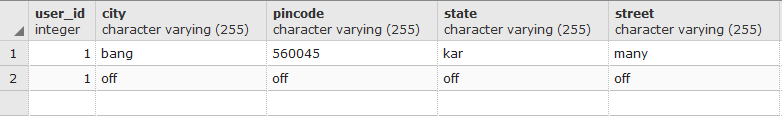
session.save(userDetails);

session.getTransaction().commit();

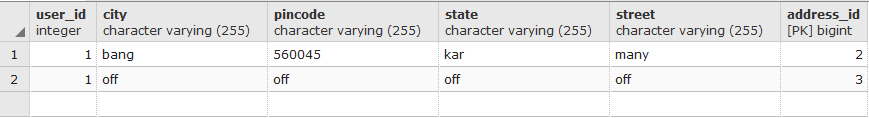
session.close();

}

ADDRESS\_ID isn’t generated **----(1)**



ADDRESS\_ID is generated **----(2)**



**Proxy Objects and Lazy/Eager Fetch Types**

@ElementCollection **//By default it is Lazy initializing**

**private** Collection<Address> list = **new** ArrayList<Address>();

@ElementCollection(fetch =FetchType.***EAGER***) **//Now it is Eager initializing**

**private** Collection<Address> list = **new** ArrayList<Address>();

When we use:

userDetails = session.get(UserDetails.**class**, 1); All the first level elements are get from the database and no the collection elements. If we manually get the collection element like: userDetails.getList().size(), then it will hit the database.

**session.get()/ session.load()**

When we do session.get(), it will always hit the database for the first level fields.

But, when we do session.load(), it will not hit the database not even for first level fields until you want to access the fields manually.

**One-TO-One**

**@JoinColumn** tells that In UserDetails there is foreign key of Vechicle **@JoinColumn** also changes the name from UserDetails\_vehicleId to vehicleId

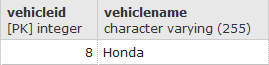
@Entity

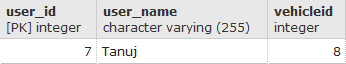
@Data  
public class UserDetails {  
  
 @Id  
 @GeneratedValue  
 private int userId;  
  
 @Column(name = "USER\_NAME")  
 private String userName;  
  
 @OneToOne  
 @JoinColumn(name="vehicleId")  
 private Vehicle vehicle;  
  
}

@Entity

@Data  
public class Vehicle {  
 @Id @GeneratedValue  
 private int vehicleId;  
 private String vehicleName;  
}

public static void main(String[] args) {  
 UserDetails userDetails = new UserDetails();  
 userDetails.setUserName("Tanuj");  
  
 Vehicle vehicle = new Vehicle();  
 vehicle.setVehicleName("Honda");  
  
 userDetails.setVehicle(vehicle);  
  
 SessionFactory sf = new Configuration().configure().buildSessionFactory();  
 Session session = sf.openSession();  
 session.beginTransaction();  
 session.save(userDetails);  
 session.save(vehicle);  
 session.getTransaction().commit();  
 session.close();  
 }





**One-To-Many**

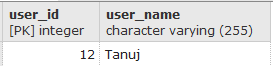
@Entity  
public class UserDetails {  
  
 @Id  
 @GeneratedValue  
 private int userId;  
 private String userName;  
 private String userName;  
  
 @OneToMany  
 private Collection<Vehicle> vehicle = new ArrayList<Vehicle>();  
   
 //or

@OneToMany   
 @JoinTable(name = "USER\_VEHICLE",  
 joinColumns = @JoinColumn(name = "USER\_ID"),   
 inverseJoinColumns = @JoinColumn(name = "VEHICLE\_ID"))  
 private Collection<Vehicle> vehicle = new ArrayList<Vehicle>();  
// The above will change the table name from userdetails\_vehicle to USER\_VEHICLE and fields to USER\_ID and VEHICLE\_ID respectively.  
  
}

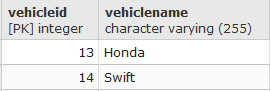
@Entity  
public class Vehicle {  
  
 @Id @GeneratedValue  
 private int vehicleId;  
 private String vehicleName;}

public static void main(String[] args){  
 UserDetails userDetails=new UserDetails();  
 userDetails.setUserName("Tanuj");  
  
 Vehicle vehicle1=new Vehicle();  
 vehicle1.setVehicleName("Honda");  
  
 Vehicle vehicle2=new Vehicle();  
 vehicle2.setVehicleName("Swift");  
  
 userDetails.getVehicle().add(vehicle1);  
 userDetails.getVehicle().add(vehicle2);  
  
 SessionFactory sf=new Configuration().configure().buildSessionFactory();  
 Session session=sf.openSession();  
 session.beginTransaction();  
 session.save(userDetails);  
 session.save(vehicle1);  
 session.save(vehicle2);  
 // Here we are saving user object as well as all the vehicle objects.  
 // We use cascading to get rid of it.  
 session.getTransaction().commit();  
 session.close();  
  
 }

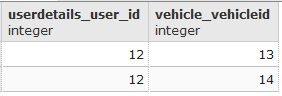
**userdetails**

****

**vehicle**

****

**Userdetails\_vehicle**

****

So, in the above 3 tables have been created i.e, One for UserDetails Entity, One for Vehicle Entity, and One for UserDetails\_Vehicle Mapping.

If we don’t want to create 3 separate table and want to show the mapping of vehicle and usertable in a single table then we have to use mappedBy under @OneToMany.

@Entity

**public** **class** UserDetails {

@Id

@GeneratedValue

**private** **int** userId;

**private** String userName;

**private** String userName;

**@OneToMany(mappedBy="user") // mappedBy=”user”**

**private** Collection<Vehicle> vehicle = **new** ArrayList<Vehicle>();

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@Entity

**public** **class** Vehicle {

@Id @GeneratedValue

**private** **int** vehicleId;

**private** String vehicleName;}

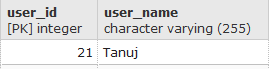
@ManyToOne

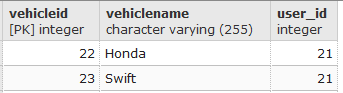
@JoinColumn(name="USER\_ID")

**private** UserDetails user;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*





***Eg2 OneToMany (Bidirectional)***

Worker can have multiple emails

@JoinColumn is applied on owner Entity(Email) where it represents foreign key that maps to Worker Entity’s primary key.

@mappedBy is applied on non-owner Entity(Woker) that enables bidirectional flow.

@Data  
@Entity  
public class Worker {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*AUTO*)  
 @Column(name = "worker\_id")  
 private int workerId;  
  
 @Column(name = "worker\_name")  
 private String workerName;  
  
 @OneToMany(mappedBy = "worker", cascade = {CascadeType.*PERSIST*,

CascadeType.*DETACH*,

CascadeType.*MERGE*,

CascadeType.*REFRESH*})  
 private List<Email> emails;  
}

@Data  
@Entity  
public class Email {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.*AUTO*)  
 private int emailId;  
  
 @Column(name = "email")  
 private String email;  
  
 @ManyToOne(cascade = { CascadeType.*PERSIST*,

CascadeType.*DETACH*,

CascadeType.*MERGE*,

CascadeType.*REFRESH*})  
 @JoinColumn(name = "worker\_id")  
 private Worker worker;  
  
}

@RestController  
public class OneToManyController {  
  
 @Autowired  
 private MyOneToManyRepository myRepository;  
  
  
 @GetMapping("/saveOneToMany")  
 public void saveIntoDataBase() {  
  
 Worker worker = new Worker();  
  
 Email email1 = new Email();  
 email1.setEmail("t@gmail.com");  
 email1.setWorker(worker);  
  
 Email email2 = new Email();  
 email2.setEmail("t@gmail.com");  
 email2.setWorker(worker);  
  
 List<Email> emailList = new ArrayList<>();  
 emailList.add(email1);  
 emailList.add(email2);  
  
 worker.setWorkerName("Tanuj");  
 worker.setEmails(emailList);  
  
 myRepository.save(worker);  
 }  
}

worker



email



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***Cascading:***

Suppose there is a One to many relationship i.e, a user can have multiple vehicles. Now in the above senerio we would have to save the user object as well as vehicle object into the database. Now if there are 100 vehicles then 100 session.save(vehicle) would have to be done and that will result into the boiler plate code. Hence we use cascading in this case as we will save only user object and all the vehicle objects will be saved automatically.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@Entity

**public** **class** UserDetails {

@Id

**private** **int** userId;

**private** String userName;

**@OneToMany(cascade = CascadeType.*PERSIST*)**

**private** Collection<Vehicle> vehicle = **new** ArrayList<Vehicle>();}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*@Entity

**public** **class** Vehicle {

@Id

**private** **int** vehicleId;

**private** String vehicleName;}

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

UserDetails user = **new** UserDetails();

user1.setUserName("Tanuj");

Vehicle vehicle1 = **new** Vehicle();

vehicle1.setVehicleName("Honda");

Vehicle vehicle2 = **new** Vehicle();

vehicle2.setVehicleName("Swift");

user.getVehicle().add(vehicle1);

user.getVehicle().add(vehicle2);

SessionFactory sf = **new** Configuration().configure().buildSessionFactory();

Session session = sf.openSession();

session.beginTransaction();

**session.persist(user**); **// Here, we are saving only user**

session.getTransaction().commit();

session.close();

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Inheritance:**

By default hibernate uses single table heirarchy. It means that hibernate creates one single table for all the objects that are inheriting a class along with class table itself.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@Entity

**public** **class** Vehicle {

@Id

@GeneratedValue

**private** **int** vehicleId;

**private** String vehicleName;}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@Entity

**public** **class** TwoWheeler **extends** Vehicle {

**private** String steeringHandle;}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@Entity

**public** **class** FourWheeler **extends** Vehicle {

**private** String steeringWheel;}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

Vehicle vehicle = **new** Vehicle();

vehicle.setVehicleName("Car");

TwoWheeler bike = **new** TwoWheeler();

bike.setVehicleName("bike");

bike.setSteeringHandle("bike steerinh handle");

FourWheeler car = **new** FourWheeler();

car.setVehicleName("car");

car.setSteeringWheel("car steering wheel");

SessionFactory sf = **new** Configuration().configure().buildSessionFactory();

Session session = sf.openSession();

session.beginTransaction();

session.save(vehicle);

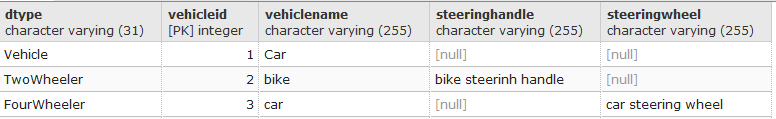
session.save(bike);

session.save(car);

session.getTransaction().commit();

session.close();}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*



\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***Same above program with some modification highlighted in yellow.***

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@Entity

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

**@DiscriminatorColumn(name = "VEHICLE\_TYPE")**

**public** **class** Vehicle {

@Id

@GeneratedValue

**private** **int** vehicleId;

**private** String vehicleName;}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@Entity

**@DiscriminatorValue("Bike")**

**public** **class** TwoWheeler **extends** Vehicle {

**private** String steeringHandle;}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

@Entity

**@DiscriminatorValue("Car")**

**public** **class** FourWheeler **extends** Vehicle {

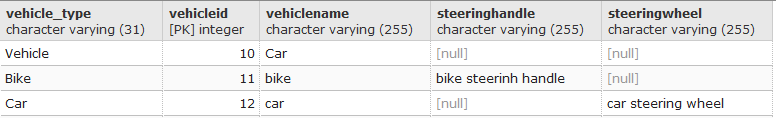
**private** String steeringWheel;}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

***HibernateTest.java***

***SAME AS ABOVE***

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

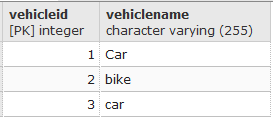
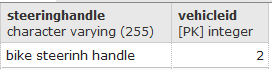


\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**If we use @Inheritance(strategy = InheritanceType.*TABLE\_PER\_CLASS*) then instead of the single table as shown above, it creates separate table for each entity.**

**If we use @Inheritance(strategy = InheritanceType.*JOINED*) : we can use join sql query in this**

**vehicle twowheeler**

** **

**fourwheeler**

****

**Cache**