

Test class coding and its execution flow

==================================

1. Create a configuration object

2. Load .cfg.xml file into configuration object using configure().

3. Build SessionFactory object using cfg which handles

a. Loading driver class

b. Creating connection

c. Prepare statement objects.

4. use SessionFactory and get Session object to perform Persistence operation.

5. Begin Transacion, if the operation in Non-Select.

6. Now perform operation using Session object.

7. Commit or rollback if transaction has started.

8. close the session at the end.

Note: To specify the configuration details and mapping details we need to write xml file.

if the filename is hibernate.cfg.xml then it promotes automatic loading, otherwise we need to read those data from "FileInputStream".

1. Using hibernate persistence operations can be peformed using methods as shown below

a. insert query

session.save(,)

session.persist(,)

b. select query

session.load(,) => It promotes lazy loading, meaning object will be created and the values will be injected. If record doesn't exists then object won't be created.

if the record is not available it would return "ObjectNotFoundException".

session.get(,) => It promotes eager loading, meaning dummy object will be created whether record exists or not with default values only when we use the Object, it will try to pull the values and keep into the Object.

If the record doesnt exists, it would return null.

c. session.update(,)

session.saveOrUpdate(,)=> first performed selection, record found, so latest values it updated using update query.

=> first performed seelction, record not found, so perform insert operation.

d. deleteQuery

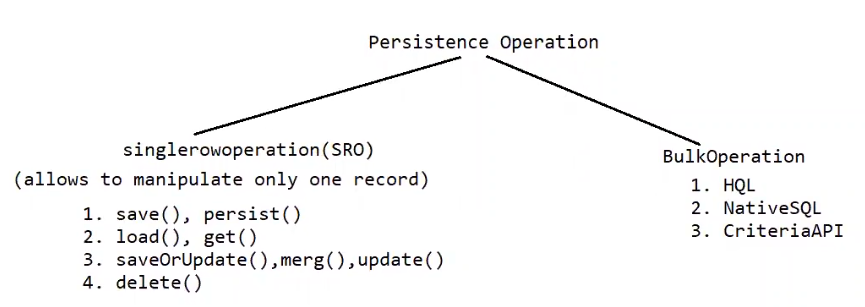
session.delete(,) : Check whether record exists, only if it exists perform deletion.

Eg: HibernateSaveOperation

Eg: SelectOperationUsingLoadMethod

Eg: DeleteOperationUsingDeleteMethod

Eg: SelectOperationUsingGetMethod



save()

=> Serializable .save(Object obj)

=> This method gives instructions to save object and also return the assigned or generated identity value back to the application as the return value.

=> This method is own method of hibernate(not per specification of JPA).

note: if generators are not configure, then value assigned to id property will be returned as identity value.

eg: increment, sequence, hilo, ......

Eg: HibernateSaveOperation

Employee.java

============

@ld

@Column(name = "eid")

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Integer empld;

As noticed above we have told hibernate to generate the value of empld, so the generated value is "Autolncrement" for MySQLDB

persist()

=>void persist(Object object)

=> return type is void, cannot return the identity value.

=> This method is given by JPA specification and it is implemented by Hibernate.

=> Gives instruction to hibernate to perform save operation on the object.

=> persist() does not allows to work with generators.

Eg: HibernatePresistOperation

Performing loading operation

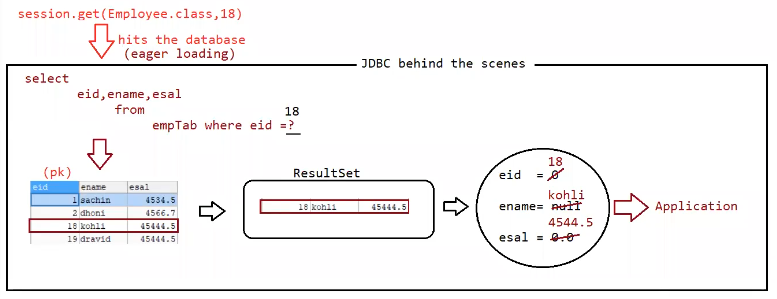
========================

get()

It perform eager loading.(hits the database and gets the record from dbtable and stores in Entity class object irrespective of whether we use that Object/not)

if we call get(), automatically the hibernate will generate the sqlquery and hits the database.

even if the record is not available still its the database, as a result of which we say get() is costly in realtime applications.



Eg: GetOperationWithXml

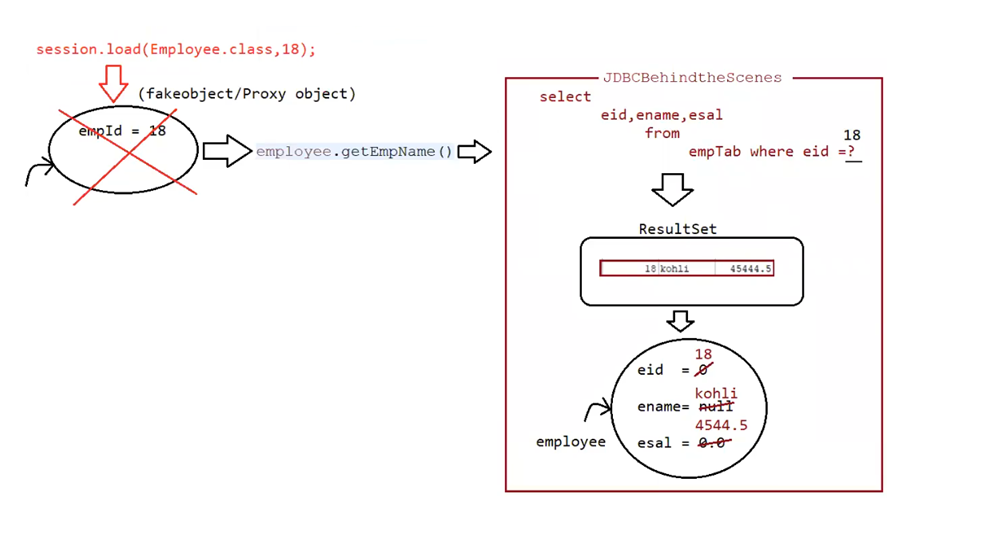
load()

It perform lazy loading(hits the database only when we use the object data other than primaary keyvalue)

Upong lazy loading, first hibernate creates the proxy object and sets only pk value to it.

when we use getter methods on non-primary keyvalue then hibernate will hit the database by executing selectquery.

if the record found then it will create a new object and injects the value to that object, otherwise it would result in "ObjectNotFoundException".



Eg: LoadOperationInternalWorking

get()

=> It supports eager loading

=> It won't generated proxy object

=> returns null if record not available

=> suitable to check whether record available or not.

=> Creates only object for Entity class.

=> Best suited for standalone applications ( guaranteed that loaded object will be used)

load()

=> It supports lazy loading

=> It generates proxy object

=> It throws ObjectNotFoundException

=> not suitable

=> Creates 2 object (proxy + Entity class)

=> Best suited for webapplications (DAO-> Service-> Controller-> View ( jsp using the object is not guaranteed)

Update Operation

1.update()

i> void update(Object object)

This method is used to modify the record of the DBTable.

Set the primary key value and change the other non-primary data for updation.

To use update(), we should remember whether record exists or not for the give primary key value. otherwise it would result in "HibernateException".

It would directly generate "update query" without "select query".

ii>Load the object from database and then modify

Here we won't get Exception as the object is available we do modify the Object.

Eg: UpdateOperationInternalWorking

2.saveOrUpdate()

If the object/record is already available only then it will update the record otherwise it will insert/create a new record

Eg: SaveOrUpdateInternalWorking

3. merge()

On the loaded object, if we want to update the data then we need to go for merge()

Eg: MergeOperationInternalWorking

Deleting object

============

=> It refers to deleting the record represented by the Entity class Object.

=> Delete the record based on the id value of the given entity object.

=> public void delete(Object obj)

Approach-1

===========

session.delete(Object obj)

directly we are trying to delete the object, so not a good approach

internally first select operation is done, if the record is available only then delete operation is performed. Otherwise only select operation is performed.

Eg: DeleteOperationInternalWorking

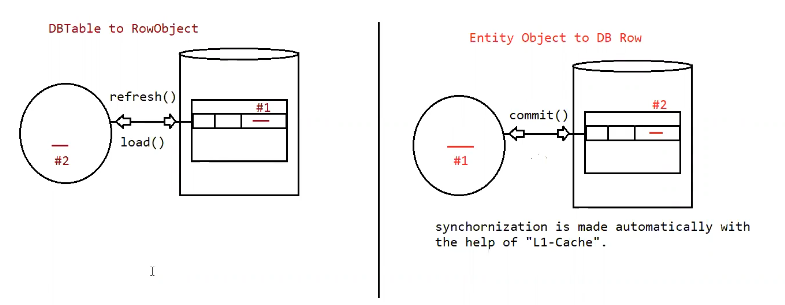
Approach-2

===========

First load the object, if found only then delete the object.

Eg: DeleteOperationWithGet

How can u show that synchronization would exists b/w EmtityObject to DBTable row?



Eg: EntityObjectToDbRow

Eg: DbTableToRowObject

Generators in hibernate

====================

MySQL => primary key value where the generation of these values are made automatic.

While creating a table, we can inform hibernate to create a columns with primary key value using @ld.

It is also possible to set the values to these primary key columns using Generators in hibernate.

There are 3 types of generators in hibernate

a. Hibernate supplied generators

b. JPA generators

c. Custom generators

Hibernate supplied generators

==========================

a. assigned

b. increment

c. identity

d. sequence

e. hilo

f. seqhilo

g. native

h. foreign

i. select

j. uuid

h. guid

assigned

=========

If we use this algorithm then explicitly we need to specify the primary key value to the table.

assigned => org.hibernate.id.Assigned

It works with all databases as we need to give the primary key value.

@ld

@Column(name = "eid")

@GenericGenerator(name="gen1",strategy = "assigned")

@GeneratedValue(generator = "gen1")

private Integer empld;

Eg: HbGeneratorsApp

increment

=========

It uses max(value) + 1 to generate the primary key value which is of int type

Works with all Database.

If dbTable is empty it will generate 1 as the identity value.

increment => org.hibernate.id.Increment

@ld

@Column(name = "eid")

@GenericGenerator(name="gen1",strategy = "increment")

@GeneratedValue(generator = "gen1")

private Integer empld;

Eg: HbGeneratorsIncrement

JPA generators

=============

These are given by Sun MS JPA specification

We will work with all ORM Frameworks

4 generators are given

a. Identity

b. sequence

c. table

d. auto

identity

======

This works in databases like MySQL but **not well** in Oracle.

It is similar to Autolncrement feature of primary key column.

Identity works with oracle only with version after oracle 12c

Oracle added support for **IDENTITY columns** starting from **Oracle 12c**, but they still have limitations (e.g., they cannot be used with batch inserts efficiently).

@ld

@Column(name = "eid")

@GeneratedValue(strategy = GenerationType.IDENTITY)

private Integer empld;

Eg: JpaGeneratorsIdentity