Hibernate Notes

What is Hibernate?

* It is a complete solution for managing persistence in Java.
* Hibernate is one of the most popular implementors of JPA specifications, provided by J2EE.
* Other implementors of JPA include iBATIS, Kodo, etc.

Advantages of Hibernate Framework over JDBC

1. Opensource & Lightweight.
2. Faster performance because internally cache is used in hibernate framework.
3. There are three types of cache in hibernate framework: first level cache, second level cache and third/query level cache.
   1. First level(L1) cache is enabled by default, cannot be disabled. First level cache is associated with Session object.
   2. Second level cache is disabled by default
   3. Third type of cache is query level cache, it caches only identifier values and results of value type. It should always be used in conjunction with the second-level cache.
4. Supports DB Independent queries: HQL (Hibernate Query Language) / JPQL (Java persistence query language). These are the object-oriented version of SQL.
5. Supports automatic persistence i.e., abstraction over JDBC.
6. Supports automatic table creation.
7. Simplifies complex join queries:

eg: To display the course names ordered by descending no of participants (many-to-many)

MySQL query -- select c.name from dac\_courses c inner join course\_studs cs on c.id = cs.c\_id inner join dac\_students s on cs.s\_id = s.stud\_id group by c.id order by count(\*) desc;

JPQL query -- select c from Course c join fetch c.students group by c.id order by count(\*) desc

1. Provides query statistics and database status: Hibernate supports Query cache and provide statistics about query and database status.
2. Hibernate translates checked SQLException to un checked org.hibernate.HibernateException. So, the programmer doesn't have to handle excs.

Advantages of using Hibernate ORM:

1. Hibernate supports Inheritance, Associations, Collections.

2. In hibernate if we save the derived class object, then its base class object will also be stored into the database, it means hibernate supporting inheritance

3. Hibernate supports relationships like One-To-Many, One-To-One, Many-To-Many-to-Many, Many-To-One

4. This will also support collections like List, Set, Map (Only new collections)

5. In JDBC all exceptions are checked exceptions, so we must write code in try, catch and throws, but in hibernate we only have Un-checked exceptions, so no need to write try, catch, or no need to write throws. Actually, in hibernate we have the translator which converts checked to Un-checked ;)

6. Hibernate has capability to generate primary keys automatically while we are storing the records into database

7. Hibernate has its own query language, i.e hibernate query language which is database independent

So, if we change the database, then also our application will work as HQL is database independent

HQL contains database independent commands

8. While we are inserting any record, if we don’t have any particular table in the database, JDBC will rises an error like View not exist, and throws exception, but in case of hibernate, if it not found any table in the database this will create the table for us ;)

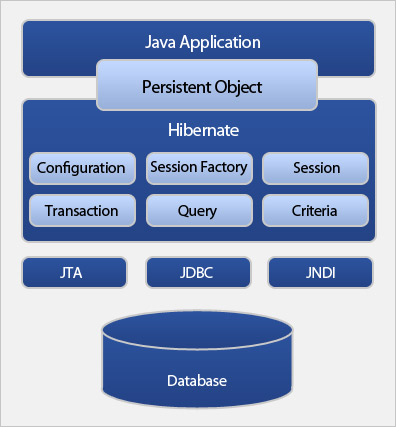
9. Hibernate supports caching mechanism by this, the number of round trips between an application and the database will be reduced, by using this caching technique an application performance will be increased automatically.

10.Hibernate supports annotations, apart from XML

11. Hibernate provided Dialect classes, so we no need to write sql queries in hibernate, instead we use the methods provided by that API.

12. Getting pagination in hibernate is quite simple.

Hibernate Architecture:



1. Configuration:

* org.hibernate.cfg.Configuration class
* The approach here is to define all configuration and mapping sources in one API and to then build the SessionFactory object in one-shot.
* All configurations are generally kept in hibernate.cfg.xml under the runtime class path.
* The configuration and mapping sources defined here are just held here until the SessionFactory is built. Making any changes to the configuration file won’t affect SessionFactory object after it has been built.

Configuration cfg = new Configuration();

cfg.configure(); // Upon calling configure methods, the configuration file and all mapping files(eg pojos) are loaded in memory.

// By default, looks at src/main/resources for 'hibernate.cfg.xml'.

// If placed at some other location (with different name), provide relative path as a parameter to configure function.

SessionFactory factory = cfg.buildSessionFactory();

Or

factory = new Configuration().configure().buildSessionFactory();

2. SessionFactory:

* org.hibernate.SessionFactory (i/f) (implementation classes are present under: hibernate core jar)
* It is a provider of session objects(via openSession() and getCurrentSession()).
* Singleton instance per application per DB.
* Immutable, hence inherently thread-safe. (Immutable meaning: Making any changes to the configuration file wont affect SessionFactory object after it has been built.)
* AutoCloseable, opens and closes DB connections on its own.
* SessionFactory is a heavy weight object, therefore should be created in the beginning phase of application deployment.

openSession vs getCurrentSession

public Session openSession() throws HibernateExc

Opens new session from SF, which has to be explicitly closed(session.close()) by programmer.

public Session getCurrentSession() throws HibernateExc

Opens new session, if one doesn't exist, otherwise continues with the existing one.

Gets automatically closed upon Tx boundary(tx.commit() / tx.rollback()) or thread over(since current session is bound to current thread --mentioned in hibernate.cfg.xml property <property name="hibernate.current\_session\_context\_class">thread</property>).

3. Session

* org.hibernate.Sesion (i/f) (implementation classes are present under: hibernate core jar)
* Represents a pooled-out database connection.
* Represents a Persistence Manager.
* Session object is associated with L1 cache, hence known as Session level cache.
* Supplies CRUD APIs (eg: save, persist, get, load, createQuery, update, delete....).
* DAO layer creates session instance as per demand (one per request for CRUD operation).
* Light weight, thread un safe.
* NO NEED for accessing the session in synchronized manner: since different third representing different client requests, will have their own session object.

Session APIs:

beginTransaction(), isOpen(), isConnected(), isDirty(), flush(), save(), saveOrUpdate(), delete(), merge(), persist(), get(), load(), update(), evict(), clear(), createQuery().

4. Transaction

org.hibernate.Transaction

Transaction interface defines a unit of work. It maintains abstraction from the transaction implementation (JTA, JDBC).

A transaction is associated with a Session and is usually initiated by a call to SharedSessionContract.beginTransaction().

5. Query

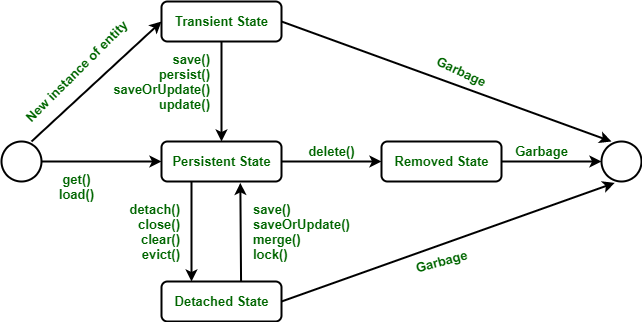
org.hibernate.query.Query (i/f)

Represents an HQL/JPQL query. Create Query object from Session object: createQuery(String queryString, Class<T> resultType)

Query API methods: getResultList(), getSingleResult(), getResultStream(), setParameter(), executeUpdate().

6. Criteria

Hibernate Lifecycle Stages



hibernate.cfg.xml properties:

1.

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

Checks if table is not yet created for a POJO, create a new table.

But if table already exists, continues with the existing table.

Other possible values: none, create, create-only, drop, create-drop, validate, update.

2.

<property name="hibernate.connection.autocommit"> false </property>

It enables auto-commit for JDBC pooled connections. However, it is not recommended.

3.

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

This property is checked by SessionFactory.getCurrentSession() Api method. It will check if there is any session bound to current thread. If yes returns existing session object otherwise returns a new session object.

Note: Hibernate cannot delete/alter the existing columns in a table, but can only append new columns.

Hibernates Automatic Dirty Checking:

The process of automatically updating the database with the changes to the persistent object when the session is flushed(@ transaction commit) is known as automatic dirty checking.

An object(POJO) enters persistent state when any one of the following happens:

When the code invokes session.save, session.persist or session.saveOrUpdate or session.merge

OR

When the code invokes session.load or session.get

OR

Result of a JPQL

Even though there is a session.flush method available but you generally don’t need to invoke it explicitly as session gets flushed when the transaction is committed.

Difference in get & load session API:

Both have a common API signature: (load/get(Class c, Serializable id)) return type is <T>

In get, if id doesn't exist = rets null

In load, if id doesn't exist & u are accessing it from within session scope, throws ObjectNotFoundException.

In get, Hibernate uses eager fetching policy ---- meaning will generate select query always & load the state from DB in persistent POJO ref. --- so even if u access the same from within the session (persistent pojo) or outside (detached) the hibernate session --- NO EXCEPTION(proxy + state)

In load, Hib uses lazy fetching policy ---- meaning it will, by default NOT generate any select query --- so what u have is ONLY PROXY (wrapper ---with no state loaded from DB) --- on such a proxy --- if u access anything outside the hibernate session(detached) ---- U WILL GET ---LazyInitializationException.

Fix (for the lazy loading of load method):

1. Change fetch type --- to eager (NOT AT ALL recommended => no caching, disabling L1 cache)

2. If u want to access any POJO in detached manner (i.e. from outside a hibernate session scope) -

Fire a non-id get method from within session & then hibernate has to load entire state from DB ---NO LazyInitializationExcption

Session APIs Guide: The instance of a mapped entity class may exist in one of three states:

Transient: Never persistent, not associated with any Session. Transient instances may be made persistent by calling save(), persist(), saveOrUpdate() or merge().

Persistent: Associated with a unique Session Persistent instances may be made transient by calling delete(). Any instance returned by a get() or load() method is persistent.

Detached: Previously persistent, not associated with any Session Detached instances may be made persistent by calling update(), saveOrUpdate(), lock() or replicate(), merge().

Note: If the Session throws an exception, the transaction must be rolled back and the session discarded.

public Transaction beginTransaction() throws HibernateException

isOpen(), isConnected(), isDirty() : Boolean returning methods.

session.save(Object object):

Returns Serializable Id. Auto persist transient instance of POJO(upon commiting the transaction), using the value of generated identifier.

session.get(Class<T> entitytype, Serializable id):

Data retrival by PK. Return the persistent instance of the given POJO with the given identifier, or null if there is no such persistent instance.

session.persist(Object object):

It makes a transient instance persistent. It also guarantees that it will not execute an INSERT statement if it is called outside of transaction boundaries.

If you assign any non-null/non-default value to id while calling persist(), it gives org.hibernate.PersistentObjectException, referring that a detached entity has been passed to persist. It takes the entity as detached because of the non-null id provided.

Summary: Only default value of Id should be passed.

While if we do the same with get(), it doesn’t give any exception and ignores the id we have passed, creates its own id and inserts a row.

JPQL Examples

Fetching multiple records(rows):

String jpql = "select c from Course c";

Query query = session.createQuery(jpql, Course.class);

List<Course> courses = query.getResultList();

Passing IN Parameters to JPQL Query:

String jpql = "select c from Course c where c.title= :title";

Course course= session.createQuery(jpql, Course.class).setParameter(“title”, title).getSingleResult();

Updating Entities:

Can be done by select followed by update(setter methods) or using update query(Bulk Update)

1st case: Update course price by course title.

String jpql = "select c from Course c where c.title= :title";

Course course= session.createQuery(jpql, Course.class).setParameter(“title”, title).getSingleResult();

Course.setFees(newFees);

2nd case: Reduce price of all courses started after a certain date.

String jpql = "update Course c set c.fees=:newFees where c.startdate<:startdate";

int updatecount= session.createQuery(jpql).setParameter(“newFees”, newFees).setParameter(“startdate”, startdate).executeUpdate();

Delete Operations:

1st case: Delete student with its PK (studentId). Use session API method: get followed by delete.

Student student = session.get(Student.class, studentId);

If(student != null){

session.delete(student);

}

2nd case: Delete student with email, use JPQL.

String jpql = “delete Student s where s.email=:email”;

int updatecount= session.createQuery(jpql).setParameter(“email”, email).executeUpdate();

session.saveOrUpdate(Object object):

Returns void.

null id -- fires insert (works as save)

non-null BUT existing id -- fires update (works as update)

non-null BUT non existing id -- throws StaleObjectStateException --to indicate that we are trying to delete or update a row that does not exist.

session.merge(Object object):

Returns persistent entity reference.

null id -- fires insert (works as save)

non-null BUT existing id – select followed by update

non-null BUT non existing id -- no exc thrown -- select followed by update, ignores users passed id

session.update(Object detachedEntity):

Returns void. Makes a detached entity reference as persistent and updates this persistence instance.

null id – throws org.hibernate.TransientObjectException

non-null BUT existing id – fires update query

non-null BUT non existing id -- throws org.hibernate.StaleStateException

Triggers for persistent to detached entity transition:

1.Session closing

2.public void evict(Object persistentEntityRef): Detach one specific entity

3.void clear(): Detach all persistent entities.

session.contains(Object entityRef): checks if in persistent state

JPA Annotations:

1. @MappedSuperclass
2. @Entity
3. @Id = Should be a serializable type (use Wrapper classes). Reason is the default values of primitives.
4. @GeneratedValue(strategy = GenerationType.IDENTITY)
5. @Table(name = “table\_name”, uniqueConstraints = @UniqueConstraint(columnNames = {“first\_name”, “last\_name”} ))
6. @Column(name = “column\_name”, length = 25, unique = true)
7. @Enumerated(EnumType.STRING/ EnumType.ORDINAL)
8. @Transient = Skip from persistence
9. @Lob = LongBlob for MySql
10. @OneToOne
11. @OneToMany(mappedBy=””, cascade = CascadeType.ALL, fetch = FetchType.LAZY, orphanRemoval = true): The value of mapped by property will be name of association. property as it appears on owning side.
12. @ManyToOne + @JoinColumn(name=””)
13. @ManyToMany
14. @CreationTimestamp = private LocalDateTime createdOn;
15. @UpdateTimestamp = private LocalDateTime updatedOn;

Default Fetch Type for AnyToMany relations is LAZY and AnyToOne is EAGER. This can be altered using fetch property.

Hibernate generated the linking table as it is unaware about the owning side and the inverse side of mapped entities. Hence to solve this problem, used mappedBy attribute on inverse side of mapped entity.

To rename the column of foreign key reference generated on the owning side of mapped entity, use @JoinColumn(name=””) annotation.

CascadeType: Denotes that the operation must be cascaded to the target of the association.

orpahanRemoval: Defines whether to apply the removal operation to entities that have been removed from the relationship and cascade the remove operation to those entities.

Examples with mapped entities (unique cases): category 1<----->\* product

1.Adding product to category –

Category category=session.get(Category.class, categoryId);

if (category != null) {

category.addProduct(newProduct); //helper method

session.persist(newProduct);

mesg="product added ID "+newProduct.getId();

}

tx.commit();

2.Delete a category – FK constraint violation exception

String jpql = "delete from Category c where c.categoryName=:nm";

int updateCount = session.createQuery(jpql).setParameter("nm", categoryName).executeUpdate();

3.Delete a category – The correct way.

String jpql = "select c from Category c where c.categoryName=:nm";

Category category = session.createQuery(jpql, Category.class).setParameter("nm", categoryName).getSingleResult();

session.delete(category);

tx.commit();

4.Orphan removal use case – here if orphan removal is not set to true, Hibernate simple delinks the entity (makes FK null), but the corresponding child record is not deleted.

Course course = session.get(Course.class, courseId);

Student student = session.get(Student.class, studentId);

if(course != null & studentId != null) {

course.removeStudent(student);

msg = "Student deleted successfully.";

}

5. OneToMany: LAZY fetching – Trying to display students enrolled in a course along with course details itself.

Case1: LazyInitializationException

String jpql = "select c from Course c where c.title=:title";

course = session.createQuery(jpql, Course.class).setParameter("title", title).getSingleResult();

tx.commit();

Case2: Change Hibernate fetch policy to EAGER (fetch=FetchType.EAGER). Not a recommended solution, will lead to worst performance.

Case3: Access the size of the collection within session scope in DAO layer. The disadvantage is that Hibernate fires multiple queries (one for each student record) to get the complete details.

Case4: Use a single join query. Also, a solution to select n+1 issue. Use ‘join fetch’ keyword of jpql.

String jpql = "select c from Course c left outer join fetch c.students where c.title=:title";

6. OneToOne – Address 1<----->1Student. An address entry can be created upon successful student registration.

7.ManyToMany -