**Hibernate**

[**https://www.youtube.com/watch?v=c2abcy0Q1a0&index=10&list=PL4AFF701184976B25**](https://www.youtube.com/watch?v=c2abcy0Q1a0&index=10&list=PL4AFF701184976B25)

* It is a ORM tool
* Used in data layer of application
* Implements JPA

Why ORM?

The Problem:

* Mapping member variables to columns
* Mapping relationships

Ex: A User Object a reference to Address object. We have to map User object to USER table and Address object to ADDRESS table. If Primary key of Address has to hold as foreign key to User table. Handling will be difficult.

* Handling data types: Ex Boolean. We don’t have Boolean data type in Database; we have to manually handle these kind of data types.
* Managing changes to Object state

To overcome all these problems, ORM came into picture.

**Object Relational Mapping.**

**Saving without Hibernate:**

* JDBC Database configuration
* The Model object
* Service method to create model object
* Database design
* DAO method to save the object using SQL queries

**Saving with Hibernate:**

* JDBC Database configuration – Hibernate Configuration
* The Model object – Annotations
* Service method to create the model object – Using the Hibernate API
* Database design – Not needed!
* DAO method to save the object using SQL queries – Not needed!

**Save using Hibernate API:**

* Create a session factory – This is only one object per application. What is does is, it creates sessions how many sessions you want it based on your application.
* Create sessions from Session Factory – Sessions are used to Save, retrieve the data or update the information
* Use those sessions to save model objects

The best way to skip some of the variables saving to DB is using **@Transient**

**@Temporal:** Is used on Date column.

If Temporal(TemporalType.DATE) is then it stores only Date

TemporalType.TIME

TemporalType.TIMESTAMP

@Lob: Large Objects. By marking as Lob, then hibernate assume it as a CLOB. If byte array it chooses as BLOB. If String it chooses CLOB.

**Natural Key:**

Email column in a table should be unique. It may be used in other table to fetch user details based on email. This kind of maintaining uniqueness (Primary key) on column is called **Natural Key**.

**Surrogate Key:**

A primary key is just used to generate serial number for a table. This kind of key is called **Surrogate Key**.

@Embedded

@Embeddable

@AttributeOverrides

@AttributeOverride

@Embedded

@AttributeOverrides({

@AttributeOverride(name=”street”, column = @Column(name = “HOME\_STREET\_NAME”)),

@AttributeOverride(name=”street”, column = @Column(name = “HOME\_CITY\_NAME”)),

})

@EmbeddedId: It is used to embed Primary key from Embeddable class.

@ElementCollection: It is used to persist the collection of Address.

@JoinTable(name = “USER\_ADDRESS”): Using above annotation it creates table with collection name whatever we defined in entity. But using @JoinTable with name attribute, we can able to create table with USER\_ADDRESS.

And reference key of UserDetails table into Address table is created with UserDetails\_userId.

--13 Chapter

@JoinTable(name = “USER\_ADDRESS”,

joinColumns=@JoinColumn(name = “USER\_ID”))

In order to have collection of Object as a member variable in your entity class,

first thing is Datatype of collection that supports Indexes.

Then @CollectionId tells that the collection defined should have an identifier.

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

I defined ADDRESS\_ID as a primary key, but what kind of generator, we should define and how?

@GenericGenerator(name=”hilo-gen”, strategy=”hilo”)

@OneToMany

@OneToOne

@ManyToMany

**Transient Object:**

If we are not passing an Object to Session.save(), this object is said to be Transient Object.

Hibernate doesn’t know, that this object has to be saved.

Before an object is handed over to Hibernate is called Transient object.

**Persistent Object:**

When I use Session.save(), this object is Persistent Object. Hibernate tracks that object and saves it.

When I do a session.save(), then I am giving responsibility to Hibernate that the object state matches the Database state.

**Detached Object:**

Detached Object is similar to the Transient object. Once session is closed, then the object will be a Detached Object. Hibernate is not going to track the changes on the Object after session is closed.

**@org.hibernate.annotations.Entity(selectBeforeUpdate = true):** This is used to Update any object into Database only, if any changes are there in the Object in the DB. If changes are there, then only it updates.

**Named Queries:**

Named queries help you to write query at entity level.

@NamedQuery(name = “UserDetails.byId”, query = “from UserDetails where userId=?”) – This is written in entity using HQL query

Query query = session.getNamedQuery(“UserDetails.byId”); -- This is in DAO

If you want to use Native SQL query in named Query:

@NamedNativeQuery(name = “UserDetails.byName”, query = “select \* from USER\_DETAILS where USER\_NAME=?”, resultClass=UserDetails.class)

Query query = session.getNamedQuery(“UserDetails.byname”);

**Hibernate Cache:**

* First Level cache – Session
* Second Level cache – Which holds data for long time

@Cacheable

@Cache(name=CacheConcurrencyStrategy.READ\_ONLY)

<property name=*"cache.use\_second\_level\_cache"*>true</property>

<property name=*"cache.provider\_class"*>org.hibernate.cache.CacheProvider</property>

**Difference between update and merge?**

**Eager Initialization:** The object is created in the program execution, in a normal way, which sometimes the programmer may not use it in the program. This looks waste of of memory and processor time.

Other way is to create the object when the programmer really requires. This is called **Lazy Initialization.**

**@OneToMany and @ManyToMany associations are defaulted to LAZY loading; and @OneToOne and @ManyToOne are defauted to EAGER loading.**

**Difference between session.get() and session.load()?**

|  |  |  |
| --- | --- | --- |
|  | Get() | Load() |
| 1 | Return null value if possible | Never returns null |
| 2 | Fast if record exists | Slow if record exists |
| 3 | Used to retrieve object | Used for delete etc. operations |
| 4 | Eager fetching | Lazy fetching |
| 5 | Always hits the DB | Not always hits the DB |
| 6 | Does not return Proxy object | Return the Proxy Object |
| 7 | As it returns null is record not exist, execution continues | It throws ObjectNotFoundException, if record not found. Execution terminates if not handled successfully. |
|  |  |  |

**Use of Inverse attribute:**

Inverse attribute tells which side of association is responsible for maintaining relationship.

It specifies that if parent is responsible for updating relationship with child while saving parent or child will update relationship itself.

If inverse = “true” in list of Child in Parent class, then child is responsible of saving the Parent Id into child object.

It is always declaring in one-to-many and many-to-many relationship.

**Use of Cascade attribute:**

Cascade is mandatory, when ever we apply relationship between objects, cascade attribute transfers operations done on one object onto its related child objects.

If we write cascade = “all”, then all the operations at parent object will be effected to child class object too.

Ex: Operations like insert, update, delete at the parent object will be effected to child object too.

Default value is “none” means no operations will be transfers to the child object.

**All-delete-orphan:** If we load one parent object from the DB then child objects related to that parent object will be loaded into one collection right.

Now if we delete the child object from that collection, then relationship between the parent object and that child object will be removed, but the record (object) in the DB will remains as it is, so if we load the same parent object again then this deleted child will not be loaded (but it will be available in DB).

So finally all-delete-orphan means, breaking the relation between objects not deleting objects from the DB.

**Difference between list() and Iterate()**

|  |  |  |
| --- | --- | --- |
|  | List() | Iterate() |
| 1 | With one db hit, all the records are loaded | For each record, one DB hit is made |
| 2 | If no cache is available, this is faster | If no cache is available, this is slower |
| 3 | Eager loading | Lazy loading |

**Fetch Types:**

Lazy fetch and Eager Fetch.

By default,

@OneToMany(fetch=FetchType.LAZY)

@Fetch(FetchMode.SELECT)

**Fetching Strategies:**

* **JOIN**: Disable the Lazy loading, always load all the collections and entities.
* **SELECT:** Lazy load all the collections and entities.
* **SUBSELECT:** Group its collections into a sub select statement.

**Version:**

Once the object is saved in a database, we can modify that object any number of times. If we want to to know how many no. of times that an object is modified, then we need to apply this Versioning concept.

Whenever you use versioning then Hibernate inserts version number as Zero, when ever the object is saved for the first time in the database. Later Hibernate increments that version no. by one automatically when ever a modification is done on that particular object.

**Caching in Hibernate:**

Hibernate caching improves the performance of the application by pooling the object in the cache.

Cache actually stores the data already loaded from the DB, so that the traffic between our application and the database will be reduced when the application wants to access that data again.

Cache stores only the data related to current running application.

**First Level Cache:**

Session object holds the first level cache data. It is enabled by default. The first level cache data will not be available to entire application. An application can use many session object.

Session object is created on demand from session factory and it is lost, once the session is closed.

Similarly, first level cache associated with session object is available only till session is alive.

**Evict():** It is used to remove particular object from cache associated with the session.

**Clear():** Used to remove all the cached objects associated with session.

**Second Level Cache:**

SessionFactory object holds the second level cache data. The data stored in the second level cache will be available to entire application. But we need to enable it manually.

**Hibernate Architecture:**

Hibernate uses various existing Java APIs, like JDBC, Java Transaction API(JPA), and Java Naming and Directory Interface (JNDI). JDBC allowing almost any database with a JDBC driver to be supported by Hibernate. JNDI and JTA allow hibernate to be integrated with J2EE application servers.

**Configuration Object:**

This is the first Hibernate object you create in Hibernate Application and usually only once created during application initialization.

Database connection: hibernate.properties and hibernate.cfg.xml

Class Mapping setup:

**SessionFactory Object:**

Configuration Object is used to create SF object which inturn configures Hibernate for the application using the supplied configuration file and allows the session object to be instantiated. This is a thread safe object.

**Session Object:**

A Session is used to get physical connection with Database. The Session object is lightweight and designed to instantiate each time an interaction is needed with DB. Persistent objects are saved and retrieved through a session object.

They are not thread safe.

**Transaction Object:**

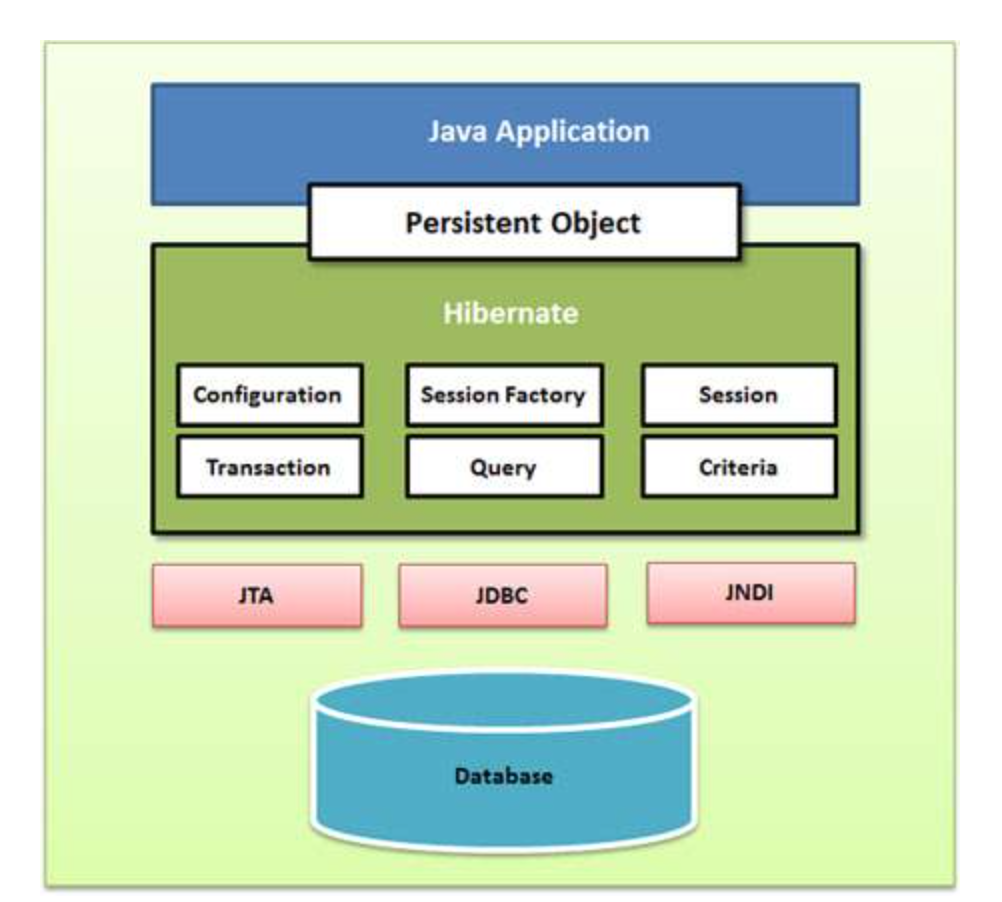
This represents a unit of work with the database and most of the RDBMS supports transaction functionality. Transactions in Hibernate are handled by an underlying transaction manager and transaction (from JDBC or JTA).

**Query Object:**

This use SQL or HQL string to retrieve data from the database and create objects. Query instance is used to bind query parameters, limit the number of results returned by the query, and finally to execute the query.

**Criteria Object:**

This objects are used to create and execute object oriented criteria queries to retrieve objects.



|  |
| --- |
| 1. Configuration cfg=**new** Configuration(); 2. cfg.configure("hibernate.cfg.xml");//populates the data of the configuration file 4. //creating seession factory object 5. SessionFactory factory=cfg.buildSessionFactory(); 7. //creating session object 8. Session session=factory.openSession(); 10. //creating transaction object 11. Transaction t=session.beginTransaction(); 13. Employee e1=**new** Employee(); 14. e1.setId(115); 15. e1.setFirstName("sonoo"); 16. e1.setLastName("jaiswal"); 18. session.persist(e1);//persisting the object 20. t.commit();//transaction is commited       session.close(); |

**Difference between update() and merge()?**

Both update() and merge() methods are used to convert the object which is in detached state into persistence state.

|  |
| --- |
| 1. SessionFactory factory = cfg.buildSessionFactory();   2) Session session1 = factory.openSession();  3) Student s1 = null;  4) Object o = session1.get(Student.class, new Integer(101));  5)s1 = (Student)o;  6)session1.close();  7) s1.setMarks(97);  8) Session session2 = factory.openSession();  9) Student s2 = null;  10) Object o1 = session2.get (Student.class, new Integer(101));  11) s2 = (Student)o1;  12) Transaction tx=session2.beginTransaction();  13) session2.merge(s1); |
|  |

From Line no: 3-6, we just loaded one object s1 into session1 cache and closed session1 at line number 6, so now object s1 in the session1 cache will be destroyed as session1 cache will expires when ever we say session1.close().

Now s1 object will be in some RAM location, not in session1 cache.

Here s1 is detached state, and at line number 7 we modified the detached object s1, now if we call update() method, then hibernate will throw an error, because we can update the object in the session only.

So we opened another session(session2) at line number 8, and again loaded same student object from the database, but with name s2.

So in this session2, we call session2.merge(s1); now into s2 object s1 changes will be merged and saved into database.

**Hibernate Session save:**

As the method name suggests, hibernate save() can be used to save entity to database. We can invoke this method outside the transaction, that’s why I don’t like this method to save data. If we use this without transaction and we have cascading between entities, then only the primary entity gets saved unless we flush the session.

* We should avoid save outside transaction boundary; otherwise mapped entities will not be saved causing data inconsistency. It’s very normal to forget flushing the session because it doesn’t throw any exception or warning.
* Hibernate save method returns the generated id immediately, this is possible because primary object is saved as soon as save method is invoked.
* If there are other objects mapped to primary object, they gets saved at the time of committing transaction or when we flush the session.
* Hibernate save load entity object to persistent context, if you will update the object properties after the save call but before the transaction is committed, it will be saved into database.

**Hibernate Persist:**

Hibernate persist is similar to save (with transaction) and it adds the entity object to the persistent context, so any further changes are tracked. If the object properties are changed before transaction is committed or session is flushed, it will also be saved into database.

Persist() method only within the boundary of a transaction, so it’s safe and take cares of any cascaded objects.

Finally, persist object doesn’t return anything so we need to use persisted object to get the generated identifier value.

**Hibernate saveOrUpdate():**

This results into insert or update queries based on the provided data. If the data is present in the database, update query is executed.

We can use saveOrUpdate () without transaction also, but again you will face the issues with mapped objects not getting saved if session is not flushed.

Hibernate saveOrUpdate adds the entity object to persistent context and track any further changes. Any further changes are saved at the time of committing transaction.

**Hibernate update:**