Hibernate ORM Library

Notes

Author	Faram Khambatta
Created	2011-02-19
Last updated	2017-10-12

General

- Always give an alias for table POJOs and prepend property names with alias, even for single POJO queries because hibernate may do multi table fetches in the background and not prepending the alias may cause sql column name errors.
- Execute sql queries within hibernate as

  Query q = session.**createSQLQuery(**“sql query”**)**;

Hibernate in NetBeans

Hibernate Config Wizard

Right-click ‘Source Packages’ in Projects tab and select Hibernate -> Hibernate config wizard.

- New Database Connection

    - New Driver
        - Add postgresql-9.1-902.jar
        - Give arbitrary name ‘Postgresql9’.
    - Enter host, port, db, user, password (check ‘Remember Password’)
    - Press ‘Test Connection’ to test connection.
    - JDBC URL should be like
      jdbc:postgresql://localhost:5432/testdb
    - Select Schema = public

This will create a file called *hibernate.cfg.xml*. This will be used by other NetBeans Hibernate related wizards. It is <u>not</u> used by user’s application. If Hibernate is used with Spring, a Spring config file *hibernate-config.xml* can be created which contains Hibernate related stuff and which is imported by *applicationConfig.xml*.

Hibernate Reverse Engineering Wizard

- Add all tables.
- This creates file *hibernate.reveng.xml*.

Hibernate Mapping Files & Pojos from db

- Check ‘JDK5 language features’.
- Choose package.
- Hibernate pojos and *.hbm.xml* mapping files will be created and placed in given package.

To run HQL Query in NetBeans

Right-click *hibernate.cfg.xml* and select ‘Run HQL Query’.

Hibernate with Spring

General

- ApplicationContext imports hibernate-config.xml which contains following bean definitions
  - dataSource contains jdbc connection details.
  - sessionFactory contains dataSource, .hbm mappings and any properties like show-sql, etc.
  - hibernateTemplate contains sessionFactory in its constructor. This bean is used by all DAO beans to do db ops.
  - DAO beans. Each bean contains a property that refers to hibernateTemplate.
- **HibernateTemplate** class has convenience methods (find, get, initialize, load, save, ...) for most db ops so its not required to use hibernate session methods directly.
  - If its required to access hibernate Session methods directly from HibernateTemplate then use its **execute()** method as

```
someDaoMethod(params) {
        return ht.execute(new HibernateCallback() {
                doInHibernate(Session sess) {
                        // session query stuff
                        return …
                }
        });
}
where ht = HibernateTemplate member of the dao class
```

Required Jars

For Hibernate (from hibernate 3.6.10 distro),

lib/hibernate3

- jpa/hibernate-jpa-api
- required/
  - slf4j-api, commons-collections, antlr, dom4j, javassist, jta.
- optional/c3p0/c3p0 (for pooled JDBC connections)

If using Postgresql db, also required is postgresql-9.1-902.jdbc4.jar from Postgresql website.

For Spring,

dist/spring.jar (from spring-2.5.6sec02 distro)

commons-logging
log4j

Fetching

- By default, all member entities and collections are lazily fetched.
    - In case of member entities, only its ID is fetched.
- To Eager Fetch an entity/collection, set attribute *lazy = "false"* for that entity/collection in *.hbm.xml* file.
    - Alternatively, to do it programmatically in DAO,
        getHibernateTemplate().initialize(proxy);
        or
        Hibernate.initialize(proxy);
- Within a Hibernate Session, if a member (other than ID) of a member entity is referred to or a member collection is iterated over then they are automatically initialized and returned along with parent entity by DAO.

Session.load() and entity classname mangling

- Session.load() always returns a proxy from cache. It doesn't hit the database. It changes the type of the returned entity.
  e.g. Person might become Person_$$_javassist_0.
- This is not a problem if detached object is used as is because this new type is a sub type of original entity type.
- But if this retrieved type is later used in another session for another get() or load() then these methods will fail because they won't find an entity of given mangled subtype.
- In this case, *Hibernate.getClass(proxy)* will give the original class name minus the mangling.

Get ID of persistent entity

session.getIdentifier(entity);

Hibernate Query Language (HQL)

Comparing timestamp/dates

If table contains a timestamp (millisecs since epoch) field then in corresponding pojo class its type is Date. If its needed to compare the timestamps then use

query.**setTimestamp**() to add the query params like

```
Query query = session.createQuery("from Pojo p where p.date between ? and ? ");
query.setTimestamp(0, from);// from is of type Date
query.setTimestamp(1, to);              // to is of type Date
```

If its required just to compare the date parts of the timestamps then use query.**setDate**() instead of query.setTimestamp().

Using lists in where clause

To use a java list in a where clause use query.**setParameterList()** like

```
Query query = session.createQuery("from Pojo p where p.person in (:persons) ");
query.setParameterList("persons", persons);// persons is of type List<Person>
```

Where clause in Delete statements

A subselect is needed in a delete query if where clause contains properties of a child entity because hibernate doesn't allow properties of child entities in the where clause of a delete query.

```
sn.createQuery("delete options o where o.addr in  ( from Addr a where a.key = :key)")
```

Transactions

```
Transaction tx = session.beginTransaction();
try {
        // do stuff
        tx.commit();
} catch(Exception e) {
        tx.rollback();
}
```

Joins

Item

ITEM_ID	DESC
1	item1
2	item2
3	item3

Bid

BID_ID	ITEM_ID	AMOUNT
1	1	99
2	1	100
3	1	101
4	2	4

Implicit Association Join (Inner join)

```
FROM Bid b WHERE b.item.desc = 'item1'
```

This join is always directed along,

Many → 1 association

1 → 1 association

It cannot be applied to,

1 → Many association

e.g. `item.bids.amount` is not valid.

Inner Join (declared explicitly in From clause)

```
FROM Item i JOIN i.bids b
```

```
WHERE i.desc = 'item1'
AND b.amount > 100
```

Above query returns list of `Object[]` pairs in which first element of arry is of type *Item* and second element is of type *Bid*.

If only a subset of objects are required then use *Select* clause as usual,

```
SELECT i FROM Item i JOIN i.bids b …
```

Inner Join Fetch

```
SELECT i FROM Item i JOIN FETCH i.bids
WHERE i.desc = 'item1'
```

In above case, *bids* collection is initialized (overriding Hibernate's default lazy fetch) when its parent *Item* object is returned by the query.

Outer Join

```
FROM Item i LEFT JOIN i.bids b
WITH b.amount > 100
WHERE i.desc = 'item1'
```

Returns *all* items where *desc* = *'item1'* and also returns bids (for those items that have bids) which have *amount* > *100*.

Note: ***with*** condition on *bids* doesn't affect condition on *Items*.

Outer Fetch Join

```
FROM Item i
LEFT JOIN FETCH i.bids
WHERE i.desc = 'item1'
```

In above case, *bids* collection cannot be restricted using *with* clause. All collections must be fully initialized.

bids collection can be restricted using *where* clause but then the restriction will apply to full query, not just the collection/association.

Fetch Join Caveats

- Fetch joined associations/collections cannot be used in restrictions (*where* clause) or projections (*select* clause).
- Don't fetch more than one collection in parallel otherwise a cartesian product is created (which is expensive).

If a collection is eager fetched, duplicates may be returned. To filter out duplicates, use *distinct* keyword,

```
SELECT DISTINCT i FROM Item i JOIN FETCH i.bids
```

This filtering of duplicates only works at the Hibernate level. The duplicates cannot be avoided in the underlying sql.

e.g. Sql for above query will return,

ITEM_ID	DESC	BID_ID	ITEM_ID	AMOUNT
1	item1	1	1	99
1	item1	2	1	100
1	item1	3	1	101
2	item2	4	2	4

Using *distinct* will cause only one ITEM_ID = 1 to be returned along with its collection of 3 bids.

Theta style Joins

```
FROM User u, LogRecord l
WHERE u.username = l.username
```

It is useful where classes don’t know anything about each other.

Criteria Queries

This style of query is useful when a query needs to be dynamically generated based on many optional parameters (e.g. from a web form).
In contrast, HQL queries are easier to understand when using predefined queries.

```
// Create a criteria query on a source class
Criteria crit = session.createCriteria(Item.class);

// return list of all entities of given (Item) type
return crit.list();

// add a restriction on a field
crit.add(Restrictions.eq(“email”, “xyz@abc.com”);

// add a restriction on an association (collection or single-valued).
// Aliases can be nested using . operator
crit.createAlias(“bids”, “b”);
crit.add(Restrictions.ge(“b.amount”, 100));

// Combining restrictions.
// By default, adding restrictions to a criteria ANDs them.
// To OR restrictions,
crit.add(Restrictions.or(Restrictions.and(..., …), …);

// eager fetch associations that are part of restrictions
// Note - this is a workaround (from here) and not encouraged by Hibernate.
// Aliases can be nested using . operator
// Note - only the associations that pass the restriction will be fetched. This may not be what is intended.
crit.createAlias("bids", "b", Criteria.LEFT_JOIN);
crit.setFetchMode("bids", FetchMode.SELECT);
crit.add(Restrictions.ge(“b.amount”, 100));
```

```
// To remove duplicate entities (due to outer joins)
crit.setResultTransformer(Criteria.DISTINCT_ROOT_ENTITY);

// Order by
crit.addOrder(Order.desc(“date”));

// Set limit
crit.setMaxResults(123);
```

Distinct, Limit, Offset, Order, Fetch associations

If it’s required to -

1. Get a Distinct list of entities
2. With conditions on fields of the entity and its associations
3. Ordered by immediate fields of the entity or 1:1 associations
4. Offset and Limit the result set.
5. Eagerly Fetch associations.

A 3-step process should be used -

1. Write a DetachedCriteria subquery that gets **distinct** (at SQL level) list of top-level entity **IDs** that fulfill all conditions.
2. Write a Criteria query whose entities have IDs **in** the list of IDs returned by previous subquery. The entities will have **order, limit** and **offset**.
3. Fetch required associations using *Hibernate.initialize().*

e.g.

```
        // DetachedCriteria subquery returns list of Distinct Person IDs that satisfies given conditions.
        DetachedCriteria dc = DetachedCriteria.forClass(Person.class);
        dc.setProjection(Projections.distinct(Projections.id()));

        if (age != null) {
            dc.add(Restrictions.le("minAge", age));
            dc.add(Restrictions.ge("maxAge", age));
        }

        // Add other Restrictions as required
```

```
// Criteria query returns Ordered Offsetted Limited list of Person, based on list of Distinct IDs from previous criteria.
Criteria crit = session.createCriteria(Person.class);
// These aliases for 1:1 associations are added to allow sorting by their immediate (primitive) members.
crit.createAlias("address", "address", Criteria.LEFT_JOIN);

// Add subquery to query
crit.add(Property.forName("id").in(dc));

// Order by any immediate member of top-level entity or immediate member of 1:1 association.
if (order != null) {
    crit.addOrder(order);
}

if (limit != null) {
    crit.setMaxResults(limit);
}

if (offset != null) {
    crit.setFirstResult(offset);
}

List<Person> persons = crit.list().

// Initialize Person associations using Hibernate.initialize().
for (Person p : persons) {
    Hibernate.initialize(p.getCars());
    …
}

// ‘persons’ contains ordered list of distinct persons (along with limit and offset) and associations
// that satisfies all conditions.
```

Programmatically setting log level

If it’s required to see the underlying SQL of a single Hibernate query as well as the bound parameters of that query, without wanting it for all the queries in the application, just

before executing the query, set the following Hibernate loggers to the given log levels -

```
Logger.getLogger("org.hibernate.SQL").setLevel(Level.DEBUG);
Logger.getLogger("org.hibernate.type").setLevel(Level.TRACE);
```

After executing the query, reset the log levels to INFO -

```
Logger.getLogger("org.hibernate.SQL").setLevel(Level.INFO);
Logger.getLogger("org.hibernate.type").setLevel(Level.INFO);
```

Troubleshooting

SELECT DISTINCT, ORDER BY expressions must appear in select list.

Assume there’s a person with multiple addresses and each Address refers to Person with an fk_person column.

To get Person objects from a query on Address table, in select phrase, use alias of Person directly rather than indirectly via Address table.

```
// Use following option -
select distinct pers from Address addr
join addr.Person pers
where …

// Don’t use this -
select distinct addr.Person
join addr.Person pers
where …
```

because in the latter case, on conversion to sql, Hibernate will create 2 aliases for Person associations, one for addr.Person and the other for the explicit join using 'pers' alias. This won't usually cause problems but as soon as an order by personID is done on the distinct select then query will fail because Hibernate will use the first Person alias for the columns of the select phrase and it will use 2nd alias for the where statement components and also for the order by clause. This will cause the query to fail saying - SELECT DISTINCT, ORDER BY expressions must appear in select list.