Hibernate Architecture



**SessionFactory** : (org.hibernate.SessionFactory)

A thread-safe, immutable cache of compiled mappings for a single database. A factory fororg.hibernate.Session instances. A client of org.hibernate.connection.ConnectionProvider. Optionally maintains a second level cache of data that is reusable between transactions at a process or cluster level.

**Session**: (org.hibernate.Session)

A single-threaded, short-lived object representing a conversation between the application and the persistent store. Wraps a JDBC java.sql.Connection. Factory for org.hibernate.Transaction. Maintains a first level cache of persistent the application's persistent objects and collections; this cache is used when navigating the object graph or looking up objects by identifier.

**Persistent objects and collections:**

Short-lived, single threaded objects containing persistent state and business function. These can be ordinary JavaBeans/POJOs. They are associated with exactly one org.hibernate.Session. Once theorg.hibernate.Session is closed, they will be detached and free to use in any application layer.

**Transaction**: (org.hibernate.Transaction)

(Optional) A single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC, JTA or CORBA transaction. A org.hibernate.Session might span several org.hibernate.Transactions in some cases. However, transaction demarcation, either using the underlying API or org.hibernate.Transaction, is never optional.

**ConnectionProvider**: (org.hibernate.connection.ConnectionProvider)

(Optional) A factory for, and pool of, JDBC connections. It abstracts the application from underlying javax.sql.DataSource or java.sql.DriverManager. It is not exposed to application, but it can be extended and/or implemented by the developer.

**TransactionFactory**: (org.hibernate.TransactionFactory)

(Optional) A factory for org.hibernate.Transaction instances. It is not exposed to the application, but it can be extended and/or implemented by the developer.

**Extension Interfaces:**

Hibernate offers a range of optional extension interfaces you can implement to customize the behavior of your persistence layer. See the API documentation for details.

**Hibernate JDBC Properties**

| **Property name** | **Purpose** |
| --- | --- |
| hibernate.connection.driver\_class | *JDBC driver class* |
| hibernate.connection.url | *JDBC URL* |
| hibernate.connection.username | *database user* |
| hibernate.connection.password | *database user password* |
| hibernate.connection.pool\_size | *maximum number of pooled connections* |

C3P0 is an open source JDBC connection pool distributed along with Hibernate in the **lib** directory. Hibernate will use its org.hibernate.connection.C3P0ConnectionProvider for connection pooling if you set hibernate.c3p0.\* properties.

The following is an example **hibernate.properties** file for c3p0:

hibernate.connection.driver\_class = org.postgresql.Driver

hibernate.connection.url = jdbc:postgresql://localhost/mydatabase

hibernate.connection.username = myuser

hibernate.connection.password = secret

hibernate.c3p0.min\_size=5

hibernate.c3p0.max\_size=20

hibernate.c3p0.timeout=1800

hibernate.c3p0.max\_statements=50

hibernate.dialect = org.hibernate.dialect.PostgreSQL82Dialect

**Hibernate Configuration Properties**

| **Property name** | **Purpose** |
| --- | --- |
| hibernate.dialect | The classname of a Hibernateorg.hibernate.dialect.Dialect which allows Hibernate to generate SQL optimized for a particular relational database.  **e.g.** full.classname.of.Dialect  In most cases Hibernate will actually be able to choose the correct org.hibernate.dialect.Dialect implementation based on the JDBC metadata returned by the JDBC driver. |
| hibernate.show\_sql | Write all SQL statements to console. This is an alternative to setting the log category org.hibernate.SQL to debug.  **e.g.** true | false |
| hibernate.format\_sql | Pretty print the SQL in the log and console.  **e.g.** true | false |
| hibernate.default\_schema | Qualify unqualified table names with the given schema/tablespace in generated SQL.  **e.g.** SCHEMA\_NAME |
| hibernate.default\_catalog | Qualifies unqualified table names with the given catalog in generated SQL.  **e.g.** CATALOG\_NAME |
| hibernate.session\_factory\_name | The org.hibernate.SessionFactory will be automatically bound to this name in JNDI after it has been created.  **e.g.** jndi/composite/name |
| hibernate.max\_fetch\_depth | Sets a maximum "depth" for the outer join fetch tree for single-ended associations (one-to-one, many-to-one). A 0disables default outer join fetching.  **e.g.** recommended values between 0 and 3 |
| hibernate.default\_batch\_fetch\_size | Sets a default size for Hibernate batch fetching of associations.  **e.g.** recommended values 4, 8, 16 |
| hibernate.default\_entity\_mode | Sets a default mode for entity representation for all sessions opened from this SessionFactory, defaults to pojo.  **e.g.** dynamic-map | pojo |
| hibernate.order\_updates | Forces Hibernate to order SQL updates by the primary key value of the items being updated. This will result in fewer transaction deadlocks in highly concurrent systems.  **e.g.** true| false |
| hibernate.generate\_statistics | If enabled, Hibernate will collect statistics useful for performance tuning.  **e.g.** true | false |
| hibernate.use\_identifier\_rollback | If enabled, generated identifier properties will be reset to default values when objects are deleted.  **e.g.** true | false |
| hibernate.use\_sql\_comments | If turned on, Hibernate will generate comments inside the SQL, for easier debugging, defaults to false.  **e.g.** true | false |
| hibernate.id.new\_generator\_mappings | Setting is relevant when using @GeneratedValue. It indicates whether or not the new IdentifierGeneratorimplementations are used forjavax.persistence.GenerationType.AUTO,javax.persistence.GenerationType.TABLE andjavax.persistence.GenerationType.SEQUENCE. Default tofalse to keep backward compatibility.  **e.g.** true | false |

**Note**

**Table 3.5. Hibernate Cache Properties**

| **Property name** | **Purpose** |
| --- | --- |
| hibernate.cache.provider\_class | The classname of a custom CacheProvider.  **e.g.** classname.of.CacheProvider |
| hibernate.cache.use\_minimal\_puts | Optimizes second-level cache operation to minimize writes, at the cost of more frequent reads. This setting is most useful for clustered caches and, in Hibernate, is enabled by default for clustered cache implementations.  **e.g.** true|false |
| hibernate.cache.use\_query\_cache | Enables the query cache. Individual queries still have to be set cachable.  **e.g.** true|false |
| hibernate.cache.use\_second\_level\_cache | Can be used to completely disable the second level cache, which is enabled by default for classes which specify a <cache> mapping.  **e.g.** true|false |
| hibernate.cache.query\_cache\_factory | The classname of a custom QueryCache interface, defaults to the built-in StandardQueryCache.  **e.g.** classname.of.QueryCache |
| hibernate.cache.region\_prefix | A prefix to use for second-level cache region names.  **e.g.** prefix |
| hibernate.cache.use\_structured\_entries | Forces Hibernate to store data in the second-level cache in a more human-friendly format.  **e.g.** true|false |
| hibernate.cache.default\_cache\_concurrency\_strategy | Setting used to give the name of the defaultorg.hibernate.annotations.CacheConcurrencyStrategyto use when either @Cacheable or @Cache is used.@Cache(strategy="..") is used to override this default. |

**3.4.1. SQL Dialects**

Always set the hibernate.dialect property to the correct org.hibernate.dialect.Dialect subclass for your database. If you specify a dialect, Hibernate will use sensible defaults for some of the other properties listed above. This means that you will not have to specify them manually.

**Table 3.8. Hibernate SQL Dialects (hibernate.dialect)**

| **RDBMS** | **Dialect** |
| --- | --- |
|  |  |
| MySQL5 | org.hibernate.dialect.MySQL5Dialect |
| MySQL5 with InnoDB | org.hibernate.dialect.MySQL5InnoDBDialect |
| MySQL with MyISAM | org.hibernate.dialect.MySQLMyISAMDialect |
| Oracle (any version) | org.hibernate.dialect.OracleDialect |
| Oracle 9i | org.hibernate.dialect.Oracle9iDialect |
| Oracle 10g | org.hibernate.dialect.Oracle10gDialect |
| Oracle 11g | org.hibernate.dialect.Oracle10gDialect |

## XML configuration file

An alternative approach to configuration is to specify a full configuration in a file namedhibernate.cfg.xml. This file can be used as a replacement for the hibernate.properties file or, if both are present, to override properties.

The XML configuration file is by default expected to be in the root of your CLASSPATH. Here is an example:

<?xml version='1.0' encoding='utf-8'?>

<!DOCTYPE hibernate-configuration PUBLIC

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

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

<hibernate-configuration>

<!-- a SessionFactory instance listed as /jndi/name -->

<session-factory

name="java:hibernate/SessionFactory">

<!-- properties -->

<property name="connection.datasource">java:/comp/env/jdbc/MyDB</property>

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

<property name="show\_sql">false</property>

<property name="transaction.factory\_class">

org.hibernate.transaction.JTATransactionFactory

</property>

<property name="jta.UserTransaction">java:comp/UserTransaction</property>

<!-- mapping files -->

<mapping resource="org/hibernate/auction/Item.hbm.xml"/>

<mapping resource="org/hibernate/auction/Bid.hbm.xml"/>

<!-- cache settings -->

<class-cache class="org.hibernate.auction.Item" usage="read-write"/>

<class-cache class="org.hibernate.auction.Bid" usage="read-only"/>

<collection-cache collection="org.hibernate.auction.Item.bids" usage="read-write"/>

</session-factory>

</hibernate-configuration>

The advantage of this approach is the externalization of the mapping file names to configuration. Thehibernate.cfg.xml is also more convenient once you have to tune the Hibernate cache. It is your choice to use either hibernate.properties or hibernate.cfg.xml. Both are equivalent, except for the above mentioned benefits of using the XML syntax.

With the XML configuration, starting Hibernate is then as simple as:

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

You can select a different XML configuration file using:

SessionFactory sf = new Configuration()

.configure("catdb.cfg.xml")

.buildSessionFactory();

**Persistent classes**

Persistent classes are classes in an application that implement the entities of the business problem (e.g. Customer and Order in an E-commerce application). The term "persistent" here means that the classes are able to be persisted, not that they are in the persistent state.

Hibernate works best if these classes follow some simple rules, also known as the Plain Old Java Object (POJO) programming model. However, none of these rules are hard requirements. Indeed, Hibernate assumes very little about the nature of your persistent objects. You can express a domain model in other ways (using trees of java.util.Map instances, for example).

**4.1. A simple POJO example**

**Example 4.1. Simple POJO representing a cat**

package eg;

import java.util.Set;

import java.util.Date;

public class Cat {

private Long id; // identifier

private Date birthdate;

private Color color;

private char sex;

private float weight;

private int litterId;

private Cat mother;

private Set kittens = new HashSet();

// addKitten not needed by Hibernate

public void addKitten(Cat kitten) {

kitten.setMother(this);

kitten.setLitterId( kittens.size() );

kittens.add(kitten);

}

}

The four main rules of persistent classes are explored in more detail in the following sections.

**4.1.1. Implement a no-argument constructor**

Cat has a no-argument constructor. All persistent classes must have a default constructor (which can be non-public) so that Hibernate can instantiate them usingjava.lang.reflect.Constructor.newInstance(). It is recommended that this constructor be defined with at least *package* visibility in order for runtime proxy generation to work properly.

**4.1.2. Provide an identifier property**

**Note**

Historically this was considered option. While still not (yet) enforced, this should be considered a deprecated feature as it will be completely required to provide a identifier property in an upcoming release.

Cat has a property named id. This property maps to the primary key column(s) of the underlying database table. The type of the identifier property can be any "basic" type (see [???](http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch04.html)). See [Section 9.4, “Components as composite identifiers”](http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch09.html#components-compositeid) for information on mapping composite (multi-column) identifiers.

**Note**

Identifiers do not necessarily need to identify column(s) in the database physically defined as a primary key. They should just identify columns that can be used to uniquely identify rows in the underlying table.

We recommend that you declare consistently-named identifier properties on persistent classes and that you use a nullable (i.e., non-primitive) type.

**4.1.3. Prefer non-final classes (semi-optional)**

A central feature of Hibernate, *proxies* (lazy loading), depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods. You can persist finalclasses that do not implement an interface with Hibernate; you will not, however, be able to use proxies for lazy association fetching which will ultimately limit your options for performance tuning. To persist afinal class which does not implement a "full" interface you must disable proxy generation. See[Example 4.2, “Disabling proxies in hbm.xml”](http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch04.html#persistent-classes-pojo-final-example-disable-proxies-xml) and [Example 4.3, “Disabling proxies in annotations”](http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch04.html#persistent-classes-pojo-final-example-disable-proxies-ann).

**Example 4.2. Disabling proxies in hbm.xml**

<class name="Cat" lazy="false"...>...</class>

**Example 4.3. Disabling proxies in annotations**

@Entity @Proxy(lazy=false) public class Cat { ... }

If the final class does implement a proper interface, you could alternatively tell Hibernate to use the interface instead when generating the proxies. See [Example 4.4, “Proxying an interface in hbm.xml”](http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch04.html#persistent-classes-pojo-final-example-proxy-interface-xml) and[Example 4.5, “Proxying an interface in annotations”](http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch04.html#persistent-classes-pojo-final-example-proxy-interface-ann).

**Example 4.4. Proxying an interface in hbm.xml**

<class name="Cat" proxy="ICat"...>...</class>

**Example 4.5. Proxying an interface in annotations**

@Entity @Proxy(proxyClass=ICat.class) public class Cat implements ICat { ... }

You should also avoid declaring public final methods as this will again limit the ability to generate*proxies* from this class. If you want to use a class with public final methods, you must explicitly disable proxying. Again, see [Example 4.2, “Disabling proxies in hbm.xml”](http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch04.html#persistent-classes-pojo-final-example-disable-proxies-xml) and [Example 4.3, “Disabling proxies in annotations”](http://docs.jboss.org/hibernate/orm/4.2/manual/en-US/html/ch04.html#persistent-classes-pojo-final-example-disable-proxies-ann).

**4.1.4. Declare accessors and mutators for persistent fields (optional)**

Cat declares accessor methods for all its persistent fields. Many other ORM tools directly persist instance variables. It is better to provide an indirection between the relational schema and internal data structures of the class. By default, Hibernate persists JavaBeans style properties and recognizes method names of the form getFoo, isFoo and setFoo. If required, you can switch to direct field access for particular properties.

Properties need *not* be declared public. Hibernate can persist a property declared with package,protected or private visibility as well.

# Hibernate Query Language

Hibernate Query Language (HQL) is an object-oriented query language, similar to SQL, but instead of operating on tables and columns, HQL works with persistent objects and their properties. HQL queries are translated by Hibernate into conventional SQL queries which in turns perform action on database.

Although you can use SQL statements directly with Hibernate using Native SQL but I would recommend to use HQL whenever possible to avoid database portability hassles, and to take advantage of Hibernate's SQL generation and caching strategies.

Keywords like SELECT , FROM and WHERE etc. are not case sensitive but properties like table and column names are case sensitive in HQL.

**FROM Clause**

You will use **FROM** clause if you want to load a complete persistent objects into memory. Following is the simple syntax of using FROM clause:

String hql = "FROM Employee";

Query query = session.createQuery(hql);

List results = query.list();

If you need to fully qualify a class name in HQL, just specify the package and class name as follows:

String hql = "FROM com.hibernatebook.criteria.Employee";

Query query = session.createQuery(hql);

List results = query.list();

**AS Clause**

The **AS** clause can be used to assign aliases to the classes in your HQL queries, specially when you have long queries. For instance, our previous simple example would be the following:

String hql = "FROM Employee AS E";

Query query = session.createQuery(hql);

List results = query.list();

The **AS** keyword is optional and you can also specify the alias directly after the class name, as follows:

String hql = "FROM Employee E";

Query query = session.createQuery(hql);

List results = query.list();

**SELECT Clause**

The **SELECT** clause provides more control over the result set than the from clause. If you want to obtain few properties of objects instead of the complete object, use the SELECT clause. Following is the simple syntax of using SELECT clause to get just first\_name field of the Employee object:

String hql = "SELECT E.firstName FROM Employee E";

Query query = session.createQuery(hql);

List results = query.list();

It is notable here that **Employee.firstName** is a property of Employee object rather than a field of the EMPLOYEE table.

**WHERE Clause**

If you want to narrow the specific objects that are returned from storage, you use the WHERE clause. Following is the simple syntax of using WHERE clause:

String hql = "FROM Employee E WHERE E.id = 10";

Query query = session.createQuery(hql);

List results = query.list();

ORDER BY Clause

To sort your HQL query's results, you will need to use the **ORDER BY** clause. You can order the results by any property on the objects in the result set either ascending (ASC) or descending (DESC). Following is the simple syntax of using

**ORDER BY clause:**

String hql = "FROM Employee E WHERE E.id > 10 ORDER BY E.salary DESC";

Query query = session.createQuery(hql);

List results = query.list();

If you wanted to sort by more than one property, you would just add the additional properties to the end of the order by clause, separated by commas as follows:

String hql = "FROM Employee E WHERE E.id > 10 " +

"ORDER BY E.firstName DESC, E.salary DESC ";

Query query = session.createQuery(hql);

List results = query.list();

**GROUP BY Clause**

This clause lets Hibernate pull information from the database and group it based on a value of an attribute and, typically, use the result to include an aggregate value. Following is the simple syntax of using GROUP BY clause:

String hql = "SELECT SUM(E.salary), E.firtName FROM Employee E " +

"GROUP BY E.firstName";

Query query = session.createQuery(hql);

List results = query.list();

Using Named Paramters

Hibernate supports named parameters in its HQL queries. This makes writing HQL queries that accept input from the user easy and you do not have to defend against SQL injection attacks. Following is the simple syntax of using named parameters:

String hql = "FROM Employee E WHERE E.id = :employee\_id";

Query query = session.createQuery(hql);

query.setParameter("employee\_id",10);

List results = query.list();

UPDATE Clause

Bulk updates are new to HQL with Hibernate 3, and deletes work differently in Hibernate 3 than they did in Hibernate 2. The Query interface now contains a method called executeUpdate() for executing HQL UPDATE or DELETE statements.

The **UPDATE** clause can be used to update one or more properties of an one or more objects. Following is the simple syntax of using UPDATE clause:

String hql = "UPDATE Employee set salary = :salary " +

"WHERE id = :employee\_id";

Query query = session.createQuery(hql);

query.setParameter("salary", 1000);

query.setParameter("employee\_id", 10);

int result = query.executeUpdate();

System.out.println("Rows affected: " + result);

DELETE Clause

The **DELETE** clause can be used to delete one or more objects. Following is the simple syntax of using DELETE clause:

String hql = "DELETE FROM Employee " +

"WHERE id = :employee\_id";

Query query = session.createQuery(hql);

query.setParameter("employee\_id", 10);

int result = query.executeUpdate();

System.out.println("Rows affected: " + result);

INSERT Clause

HQL supports **INSERT INTO** clause only where records can be inserted from one object to another object. Following is the simple syntax of using INSERT INTO clause:

String hql = "INSERT INTO Employee(firstName, lastName, salary)" +

"SELECT firstName, lastName, salary FROM old\_employee";

Query query = session.createQuery(hql);

int result = query.executeUpdate();

System.out.println("Rows affected: " + result);

Aggregate Methods

HQL supports a range of aggregate methods, similar to SQL. They work the same way in HQL as in SQL and following is the list of the available functions:

|  |  |  |
| --- | --- | --- |
| **S.N.** | **Functions** | **Description** |
| 1 | avg(property name) | The average of a property's value |
| 2 | count(property name or \*) | The number of times a property occurs in the results |
| 3 | max(property name) | The maximum value of the property values |
| 4 | min(property name) | The minimum value of the property values |
| 5 | sum(property name) | The sum total of the property values |

The **distinct** keyword only counts the unique values in the row set. The following query will return only unique count:

String hql = "SELECT count(distinct E.firstName) FROM Employee E";

Query query = session.createQuery(hql);

List results = query.list();

Pagination using Query

There are two methods of the Query interface for pagination.

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
| 1 | **Query setFirstResult(int startPosition)** This method takes an integer that represents the first row in your result set, starting with row 0. |
| 2 | **Query setMaxResults(int maxResult)** This method tells Hibernate to retrieve a fixed number **maxResults** of objects. |

Using above two methods together, we can construct a paging component in our web or Swing application. Following is the example which you can extend to fetch 10 rows at a time:

String hql = "FROM Employee";

Query query = session.createQuery(hql);

query.setFirstResult(1);

query.setMaxResults(10);

List results = query.list();

# Hibernate Criteria Queries

Hibernate provides alternate ways of manipulating objects and in turn data available in RDBMS tables. One of the methods is Criteria API which allows you to build up a criteria query object programmatically where you can apply filtration rules and logical conditions.

The Hibernate **Session** interface provides **createCriteria()** method which can be used to create a **Criteria**object that returns instances of the persistence object's class when your application executes a criteria query.

Following is the simplest example of a criteria query is one which will simply return every object that corresponds to the Employee class.

Criteria cr = session.createCriteria(Employee.class);

List results = cr.list();

**Restrictions with Criteria:**

You can use **add()** method available for **Criteria** object to add restriction for a criteria query. Following is the example to add a restriction to return the records with salary is equal to 2000:

Criteria cr = session.createCriteria(Employee.class);

cr.add(Restrictions.eq("salary", 2000));

List results = cr.list();

Following are the few more examples covering different scenarios and can be used as per requirement:

Criteria cr = session.createCriteria(Employee.class);

// To get records having salary more than 2000

cr.add(Restrictions.gt("salary", 2000));

// To get records having salary less than 2000

cr.add(Restrictions.lt("salary", 2000));

// To get records having fistName starting with zara

cr.add(Restrictions.like("firstName", "zara%"));

// Case sensitive form of the above restriction.

cr.add(Restrictions.ilike("firstName", "zara%"));

// To get records having salary in between 1000 and 2000

cr.add(Restrictions.between("salary", 1000, 2000));

// To check if the given property is null

cr.add(Restrictions.isNull("salary"));

// To check if the given property is not null

cr.add(Restrictions.isNotNull("salary"));

// To check if the given property is empty

cr.add(Restrictions.isEmpty("salary"));

// To check if the given property is not empty

cr.add(Restrictions.isNotEmpty("salary"));

You can create AND or OR conditions using LogicalExpression restrictions as follows:

Criteria cr = session.createCriteria(Employee.class);

Criterion salary = Restrictions.gt("salary", 2000);

Criterion name = Restrictions.ilike("firstNname","zara%");

// To get records matching with OR condistions

LogicalExpression orExp = Restrictions.or(salary, name);

cr.add( orExp );

// To get records matching with AND condistions

LogicalExpression andExp = Restrictions.and(salary, name);

cr.add( andExp );

List results = cr.list();

Though all the above conditions can be used directly with HQL as explained in previous tutorial.

**Pagination using Criteria:**

There are two methods of the Criteria interface for pagination.

|  |  |
| --- | --- |
| **S.N.** | **Method & Description** |
| 1 | **public Criteria setFirstResult(int firstResult)** This method takes an integer that represents the first row in your result set, starting with row 0. |
| 2 | **public Criteria setMaxResults(int maxResults)** This method tells Hibernate to retrieve a fixed number **maxResults** of objects. |

Using above two methods together, we can construct a paging component in our web or Swing application. Following is the example which you can extend to fetch 10 rows at a time:

Criteria cr = session.createCriteria(Employee.class);

cr.setFirstResult(1);

cr.setMaxResults(10);

List results = cr.list();

**Sorting the Results:**

The Criteria API provides the **org.hibernate.criterion.Order** class to sort your result set in either ascending or descending order, according to one of your object's properties. This example demonstrates how you would use the Order class to sort the result set:

Criteria cr = session.createCriteria(Employee.class);

// To get records having salary more than 2000

cr.add(Restrictions.gt("salary", 2000));

// To sort records in descening order

crit.addOrder(Order.desc("salary"));

// To sort records in ascending order

crit.addOrder(Order.asc("salary"));

List results = cr.list();

**Projections & Aggregations:**

The Criteria API provides the **org.hibernate.criterion.Projections** class which can be used to get average, maximum or minimum of the property values. The Projections class is similar to the Restrictions class in that it provides several static factory methods for obtaining **Projection** instances.

Following are the few examples covering different scenarios and can be used as per requirement:

Criteria cr = session.createCriteria(Employee.class);

// To get total row count.

cr.setProjection(Projections.rowCount());

// To get average of a property.

cr.setProjection(Projections.avg("salary"));

// To get distinct count of a property.

cr.setProjection(Projections.countDistinct("firstName"));

// To get maximum of a property.

cr.setProjection(Projections.max("salary"));

// To get minimum of a property.

cr.setProjection(Projections.min("salary"));

// To get sum of a property.

cr.setProjection(Projections.sum("salary"));

# Hibernate Native SQL

You can use native SQL to express database queries if you want to utilize database-specific features such as query hints or the CONNECT keyword in Oracle. Hibernate 3.x allows you to specify handwritten SQL, including stored procedures, for all create, update, delete, and load operations.

Your application will create a native SQL query from the session with the **createSQLQuery()** method on the Session interface.:

public SQLQuery createSQLQuery(String sqlString) throws HibernateException

After you pass a string containing the SQL query to the createSQLQuery() method, you can associate the SQL result with either an existing Hibernate entity, a join, or a scalar result using addEntity(), addJoin(), and addScalar() methods respectively.

## Scalar queries:

The most basic SQL query is to get a list of scalars (values) from one or more tables. Following is the syntax for using native SQL for scalar values:

String sql = "SELECT first\_name, salary FROM EMPLOYEE";

SQLQuery query = session.createSQLQuery(sql);

query.setResultTransformer(Criteria.ALIAS\_TO\_ENTITY\_MAP);

List results = query.list();

## Entity queries:

The above queries were all about returning scalar values, basically returning the "raw" values from the resultset. The following is the syntax to get entity objects as a whole from a native sql query via addEntity().

String sql = "SELECT \* FROM EMPLOYEE";

SQLQuery query = session.createSQLQuery(sql);

query.addEntity(Employee.class);

List results = query.list();

## Named SQL queries:

The following is the syntax to get entity objects from a native sql query via addEntity() and using named SQL query.

String sql = "SELECT \* FROM EMPLOYEE WHERE id = :employee\_id";

SQLQuery query = session.createSQLQuery(sql);

query.addEntity(Employee.class);

query.setParameter("employee\_id", 10);

List results = query.list();

# Hibernate Batch Processing

Consider a situation when you need to upload a large number of records into your database using Hibernate. Following is the code snippet to achieve this using Hibernate:

Session session = SessionFactory.openSession();

Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {

Employee employee = new Employee(.....);

session.save(employee);

}

tx.commit();

session.close();

Because by default, Hibernate will cache all the persisted objects in the session-level cache and ultimately your application would fall over with an **OutOfMemoryException** somewhere around the 50,000th row. You can resolve this problem if you are using **batch processing** with Hibernate.

Let us modify configuration file as to add **hibernate.jdbc.batch\_size** property:

<property name="hibernate.jdbc.batch\_size">

50

</property>

To use the batch processing feature, first set **hibernate.jdbc.batch\_size** as batch size to a number either at 20 or 50 depending on object size. This will tell the hibernate container that every X rows to be inserted as batch. To implement this in your code we would need to do little modification as follows:

Session session = SessionFactory.openSession();

Transaction tx = session.beginTransaction();

for ( int i=0; i<100000; i++ ) {

Employee employee = new Employee(.....);

session.save(employee);

if( i % 50 == 0 ) { // Same as the JDBC batch size

//flush a batch of inserts and release memory:

session.flush();

session.clear();

}

}

tx.commit();

session.close();

Above code will work fine for the INSERT operation, but if you are willing to make UPDATE operation then you can achieve using the following code:

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

ScrollableResults employeeCursor = session.createQuery("FROM EMPLOYEE")

.scroll();

int count = 0;

while ( employeeCursor.next() ) {

Employee employee = (Employee) employeeCursor.get(0);

employee.updateEmployee();

seession.update(employee);

if ( ++count % 50 == 0 ) {

session.flush();

session.clear();

}

}

tx.commit();

session.close();

# Hibernate Sessions

A Session is used to get a physical connection with a database. The Session object is lightweight and designed to be instantiated each time an interaction is needed with the database. Persistent objects are saved and retrieved through a Session object.

The session objects should not be kept open for a long time because they are not usually thread safe and they should be created and destroyed them as needed. The main function of the Session is to offer create, read and delete operations for instances of mapped entity classes. Instances may exist in one of the following three states at a given point in time:

* **transient:** A new instance of a a persistent class which is not associated with a Session and has no representation in the database and no identifier value is considered transient by Hibernate.
* **persistent:** You can make a transient instance persistent by associating it with a Session. A persistent instance has a representation in the database, an identifier value and is associated with a Session.
* **detached:** Once we close the Hibernate Session, the persistent instance will become a detached instance.

A Session instance is serializable if its persistent classes are serializable. A typical transaction should use the following idiom:

Session session = factory.openSession();

Transaction tx = null;

try {

tx = session.beginTransaction();

// do some work

...

tx.commit();

}

catch (Exception e) {

if (tx!=null) tx.rollback();

e.printStackTrace();

}finally {

session.close();

}

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

**Session Interface Methods:**

There are number of methods provided by the **Session** interface but I'm going to list down few important methods only, which we will use in this tutorial. You can check Hibernate documentation for a complete list of methods associated with **Session** and **SessionFactory**.

|  |  |
| --- | --- |
| **S.N.** | **Session Methods and Description** |
| 1 | **Transaction beginTransaction()**  Begin a unit of work and return the associated Transaction object. |
| 2 | **void cancelQuery()**  Cancel the execution of the current query. |
| 3 | **void clear()**  Completely clear the session. |
| 4 | **Connection close()**  End the session by releasing the JDBC connection and cleaning up. |
| 5 | **Criteria createCriteria(Class persistentClass)**  Create a new Criteria instance, for the given entity class, or a superclass of an entity class. |
| 6 | **Criteria createCriteria(String entityName)**  Create a new Criteria instance, for the given entity name. |
| 7 | **Serializable getIdentifier(Object object)**  Return the identifier value of the given entity as associated with this session. |
| 8 | **Query createFilter(Object collection, String queryString)**  Create a new instance of Query for the given collection and filter string. |
| 9 | **Query createQuery(String queryString)**  Create a new instance of Query for the given HQL query string. |
| 10 | **SQLQuery createSQLQuery(String queryString)**  Create a new instance of SQLQuery for the given SQL query string. |
| 11 | **void delete(Object object)**  Remove a persistent instance from the datastore. |
| 12 | **void delete(String entityName, Object object)**  Remove a persistent instance from the datastore. |
| 13 | **Session get(String entityName, Serializable id)**  Return the persistent instance of the given named entity with the given identifier, or null if there is no such persistent instance. |
| 14 | **SessionFactory getSessionFactory()**  Get the session factory which created this session. |
| 15 | **void refresh(Object object)**  Re-read the state of the given instance from the underlying database. |
| 16 | **Transaction getTransaction()**  Get the Transaction instance associated with this session. |
| 17 | **boolean isConnected()**  Check if the session is currently connected. |
| 18 | **boolean isDirty()**  Does this session contain any changes which must be synchronized with the database? |
| 19 | **boolean isOpen()**  Check if the session is still open. |
| 20 | **Serializable save(Object object)**  Persist the given transient instance, first assigning a generated identifier. |
| 21 | **void saveOrUpdate(Object object)**  Either save(Object) or update(Object) the given instance. |
| 22 | **void update(Object object)**  Update the persistent instance with the identifier of the given detached instance. |
| 23 | **void update(String entityName, Object object)**  Update the persistent instance with the identifier of the given detached in |