@Id @Entity

@Transient, @column,

@Temporal(.time,.date)

@Lob,@cacheable,@cache(startagey),@NamedQuery(name,query)

@embedble,@embedde

@elementcollection()

@attributeoverride, @attributeoverrides

 sessionFactory.evictEntity(entityName);//cache

**public** **class** UserDetails {

@Id @GeneratedValue(strategy=GenerationType.***AUTO***)

**private** **int** userId;

**private** String userName;

@ElementCollection(fetch=FetchType.EAGER)

@JoinTable(name="User\_Address",

joinColumns=@JoinColumn(name="USER\_ID(FK crated by hb)") )

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

@CollectionId(columns={@Column(name="Address\_Id(identifier colulumn like primary key)")},generator="hilo-gen",type=@Type(type="long"))

**private** Collection<Address> addresses = **new** ArrayList<Address>();

OneToMany

In Userdetail.java

@OneToMany

@JoinTable(name="USER\_VEHICLE", joinColumns=@JoinColumn(name="USER\_ID"),

inverseJoinColumns=@JoinColumn(name="Vehicle\_ID"))

**private** Collection<Vehicle> vehicle = **new** ArrayList<Vehicle>();

(OR)

@OneToMany(mappedBy="user")

**private** Collection<Vehicle> vehicle = **new** ArrayList<Vehicle>();

In vehicle.java

@ManyToOne

@JoinColumn(name="USER\_ID")

**private** UserDetails user;

ManyToMany

In Userdetail.java

@ManyToMany(mappedBy="userList")

**private** Collection<Vehicle> vehicle = **new** ArrayList<Vehicle>();

In vehicle.java

@ManyToMany

**private** Collection<UserDetails> userList= **new** ArrayList<UserDetails>();  
  
**Q.1 What is the difference between hibernate and jdbc ?**

Ans. 1) Hibernate is data base independent, your code will work for all ORACLE,MySQL ,SQLServer etc.   
In case of JDBC query must be data base specific.   
2) As Hibernate is set of Objects, you don’t need to learn SQL language.   
You can treat TABLE as a Object. Only Java knowledge is need.   
In case of JDBC you need to learn SQL.   
3) Don’t need Query tuning in case of Hibernate. If you use Criteria Quires in Hibernate then hibernate automatically tuned your query and return best result with performance.   
In case of JDBC you need to tune your queries.   
4) You will get benefit of Cache. Hibernate supports two level of cache, First level and 2nd level. So you can store your data into Cache for better performance.   
In case of JDBC you need to implement your java cache.   
5) Hibernate supports Query cache and It will provide the statistics about your query and database status.   
JDBC donot provide any statistics.   
6) Development fast in case of Hibernate because you don’t need to write queries   
7) No need to create any connection pool in case of Hibernate. You can use connection pool maintained by hibernate.  
In case of JDBC you need to write your own connection pool   
8) In the xml file you can see all the relations between tables in case of Hibernate. Easy readability.   
9) You can load your objects on start up using lazy=false in case of Hibernate.   
JDBC Don’t have such support.   
10) Hibernate Supports automatic versioning of rows but JDBC Not.

**What is lazy loading and how do you achieve that in hibernate?**

Lazy setting decides whether to load child objects while loading the Parent Object. You need to specify parent class.Lazy = true in hibernate mapping file. By default the lazy loading of the child objects is true. This make sure that the child objects are not loaded unless they are explicitly invoked in the application by calling getChild() method on parent. In this case hibernate issues a fresh database call to load the child when getChild() is actully called on the Parent object. But in some cases you do need to load the child objects when parent is loaded. Just make the lazy=false and hibernate will load the child when parent is loaded from the database.

Examples: Address child of User class can be made lazy if it is not required frequently. But you may need to load the Author object for Book parent whenever you deal with the book for online bookshop.

Hibernate does not support lazy initialization for detached objects. Access to a lazy association outside of the context of an open Hibernate session will result in an exception.

**3) What’s N+1 problem?**

Fetching an Object that has a relations with many objects(one to many). First get of the primary object generates the query for the primary object. They a fetch of each of relationship object generates one query. Let us look at a hospital that has many departments.

|  |  |
| --- | --- |
| 1  2  3  4 | class Hospital {      int Id;      Department[] departments} |

A query for Hospital generates one query. And then a get on each department generates N queries. That is an N+1 problem.

The advantages of N+1 query is that the query has less complexity and can have better performance if the joined objects are not needed with the original Objects first get.

Solution:  
1) pre-fetching in batches, this will reduce N+1 problem to N/K + 1 problem where  K is size of batch  
2) subselect fetching strategy  
3) disabling lazy loading

**4)What does session.refresh() do ?**  
It is possible to re-load an object and all its collections at any time, using the refresh() method. This is useful  
when database triggers are used to initialize some of the properties of the object.   
For Example - Triger on cat\_name coulmn. Trigger is updating hit\_count coulmn in the same Cat Table. When Insert data into Cat TABLE trigger update hit\_count coulmn to 1. sess.refresh() reload all the data. No need again to select call.   
sess.save(cat);   
sess.flush(); //force the SQL INSERT   
sess.refresh(cat); //re-read the state (after the trigger executes)

**5)How to Execute Stored procedure in Hibernate ?**  
Option 1:   
Connection con = null;   
try {   
con = session.connection();   
CallableStatement st = con   
.prepareCall("{call your\_sp(?,?)}");   
st.registerOutParameter(2, Types.INTEGER);   
st.setString(1, "some\_Seq");   
st.executeUpdate();   
Option 2:   
<sql-query name="selectAllEmployees\_SP" callable="true">   
<return alias="emp" class="employee">   
<return-property name="empid" column="EMP\_ID"/>   
<return-property name="name" column="EMP\_NAME"/>   
<return-property name="address" column="EMP\_ADDRESS"/>   
{ ? = call selectAllEmployees() }   
</return>   
</sql-query>   
SQLQuery sq = (SQLQuery) session.getNamedQuery("selectAllEmployees\_SP");   
List results = sq.list();

What is difference between openSession and getCurrentSession?

[Hibernate SessionFactory](https://www.journaldev.com/3522/hibernate-sessionfactory) getCurrentSession() method returns the session bound to the context. But for this to work, we need to configure it in hibernate configuration file. Since this session object belongs to the hibernate context, we don’t need to close it. Once the session factory is closed, this session object gets closed.

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

Hibernate SessionFactory openSession() method always opens a new session. We should close this session object once we are done with all the database operations. We should open a new session for each request in multi-threaded environment.

**Transient**: When an object is never persisted or associated with any session, it’s in transient state. Transient instances may be made persistent by calling save(), persist() or saveOrUpdate(). Persistent instances may be made transient by calling delete().

**Persistent**: When an object is associated with a unique session, it’s in persistent state. Any instance returned by a get() or load() method is persistent.

**Detached**: When an object is previously persistent but not associated with any session, it’s in detached state. Detached instances may be made persistent by calling update(), saveOrUpdate(), lock() or replicate(). The state of a transient or detached instance may also be made persistent as a new persistent instance by calling merge().

What is Query Cache in Hibernate?

Hibernate implements a cache region for queries resultset that integrates closely with the hibernate second-level cache.

This is an optional feature and requires additional steps in code. This is only useful for queries that are run frequently with the same parameters. First of all we need to configure below property in hibernate configuration file.

<property name="hibernate.cache.use\_query\_cache">true</property>

And in code, we need to use setCacheable(true) method of Query, quick example looks like below.

Query query = session.createQuery("from Employee");

query.setCacheable(true);

query.setCacheRegion("ALL\_EMP");

What is the dirty checking feature of Hibernate ?  
Dirty checking feature of the Hibernate allows users or developers to avoid time consuming data base write actions. This feature makes necessary updations and changes to the fields which require a change, remaining fields are left unchanged or untouched.

Dirty Checking  is one of the features of hibernate. In dirty checking, hibernate automatically detects whether    an object is modified (or) not and need to be updated. As long as the object is in persistent state i.e., bound to a particular Session(org.hibernate.Session). Hibernate monitors any changes to the objects and executes sql.  
Note:- For dirty checking to work, the object must exist in cache.

What are the collection types in Hibernate?

1.Bag 2.Set 3. List 4.Array 5.Map

**What’s the difference between load() and get()?**

|  |  |
| --- | --- |
| **load()** | **get()** |
| Only use the load() method if you are sure that the object exists. | If you are not sure that the object exists, then use one of the get()methods. |
| load() method will throw an exception if the unique id is not found in the database. | get() method will return null if the unique id is not found in the database. |
| load() just returns a proxy by default and database won’t be hit until the proxy is first invoked. | get() will hit the database immediately. |

session.lock() method of session class is used to reattach an object which has been detached earlier. This method of reattaching doesn’t check for any data synchronization in database while reattaching the object and hence may lead to lack of synchronization in data.

Objects which have been detached and are no longer associated with any persistent entities can be reattached by calling session.merge() method of session class.

Ans. session.save() : Save does an insert and will fail if the primary key is already persistent.   
session.saveOrUpdate() : saveOrUpdate does a select first to determine if it needs to do an insert or an update. Insert data if primary key not exist otherwise update data.   
session.persist() : Does the same like session.save(). But session.save() return Serializable object but session.persist() return void.         session.save() returns the generated identifier (Serializable object) and session.persist() doesn't.

For Example :

 System.out.println(session.save(question)); :  This will print the generated primary key.            
System.out.println(session.persist(question)); :  Compile time error because session.persist() return void.

Ans. Both of these methods and saveOrUpdate() method are intended for reattaching a detached object. The session.lock() method simply reattaches the object to the session without checking or updating the database on the assumption that the database in sync with the detached object.   
It is the best practice to use either session.update(..) or session.saveOrUpdate().   
Use session.lock() only if you are absolutely sure that the detached object is in sync with your detached object or if it does not matter because you will be overwriting all the columns that would have changed later on within the same transaction.   
Each interaction with the persistent store occurs in a new Session. However, the same persistent instances are reused for each interaction with the database. The application  manipulates the state of detached instances originally loaded in another Session and then "re-associates" them using Session.update() or Session.saveOrUpdate().   
// foo is an instance loaded by a previous Session   
foo.setProperty("bar");   
session = factory.openSession();   
session.saveOrUpdate(foo);   
session.flush();   
session.connection().commit();   
session.close();  
You may also call lock() instead of update() and use LockMode.READ (performing a version check, bypassing all caches) if you are sure that the object has not been modified.

7. Difference between session.saveOrUpdate() and session.merge()?

Ans. saveOrUpdate() does the following:   
1) if the object is already persistent in this session, do nothing   
2) if another object associated with the session has the same identifier, throw an exception  
3) if the object has no identifier property, save() it   
4) if the object's identifier has the value assigned to a newly instantiated object, save() it   
5) if the object is versioned (by a <version> or <timestamp>), and the version property value is the same value assigned to a newly instantiated object, save() it   
6) otherwise update() the object   
merge() is very different:   
1) if there is a persistent instance with the same identifier currently associated with the session, copy the state of the given object onto the persistent instance   
2) if there is no persistent instance currently associated with the session, try to load it from the database, or create a new persistent instance.   
3) the persistent instance is returned   
4) the given instance does not become associated with the session, it remains detached

**. What are the pros and cons of detached objects?**   
Pros:   
" When long transactions are required due to user think-time, it is the best practice to break the long transaction up into two or more transactions. You can use detached objects from the first transaction to carry data all the way up to the presentation layer. These detached objects get modified outside a transaction and later on re-attached to a new transaction via another session.   
Cons :  
" In general, working with detached objects is quite cumbersome, and better to not clutter up the session with them if possible. It is better to discard them and re-fetch them on subsequent requests. This approach is not only more portable but also more efficient because - the objects hang around in Hibernate's cache anyway.   
" Also from pure rich domain driven design perspective it is recommended to use DTOs (DataTransferObjects) and DOs (DomainObjects) to maintain the separation between Service and UI tiers.

**How would you reatach detached objects to a session when the same object has already been loaded into the session?**

You can use the session.merge() method call.

Can we make an Hibernate Entity Class final?

Yes, you can make Hibernate entity class final but remember hibernate uses proxies when we call load method; and a proxy works by replacing actual class object and being present in its place. This is only possible when proxy satisfies IS-A relation. According to Java doc, if you define a class as final it can be extended hence hibernate will nolonger be able to use proxy which will in turn limit your preformance tuning options.

What is the use of session.flush() in Hibernate?

session.flush() method forces Hibernate to synchronize the in-memory state of the Session with the database (i.e. to write changes to the database). But it does not means that the transaction has been committed. To commit the transaction you should explicitly call commit() method.

You can synchronize your database with small chunks of data using flush() instead of committing a large data at once using commit() and face the risk of getting an **Out Of Memory Exception**.

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

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

Customer customer = new Customer(.....);

session.save(customer);

if ( i % 30 == 0 ) { //flush a batch of inserts and release memory:

session.flush();

session.clear(); }}tx.commit();session.close();

How do you configure 2nd level cach in hibernate?

To configure second level cache in hibernate, you should add following configurations as listed below:

Add provider class in your hibernate configuration file i.e hibernate.cfg.xml

**Can we disable first level cache ? What should one do if we don't want an object to be cached ?**

Ans. No, We can either call evict after the object retrieval or can use separate sessions.

What is difference between sorted collection and ordered collection? Which one is better?

|  |  |  |
| --- | --- | --- |
|  | Sorted Collection | Ordered Collection |
| Sorting Level | The sorted collection is a collection that is sorted using the Java collections framework. The sorting is done in the memory of JVM that is running hibernate, soon after reading the data from the database using Java Comparator | The order collections will also sorts a collection by using the order by clause while the results are fetched from database. |
| Size/Efficiency | It should be used when elements of collection are less. | It should be used when elements of collection are huge. |

**.What is Hibernate proxy?**

In hibernate proxies are placeholder that are generated at runtime. Whenever hibernate returns an instance of any entity class, it checks weather it returns a proxy object to avoid the database hit.

In hibernate proxies are triggers the loading of real object when it just accessed at very first time.

When we use session.load(…….) to get the resultset of any entity with the help of identifier an by calling the property other than identifier, the proxy is initialized.

**41. What is transactional write-behind?**

Hibernate uses a sophisticated algorithm to determine an efficient ordering that avoids database foreign key constraint violations but is still sufficiently predictable to the user. This feature is called transactional write-behind.

**What is the use of dynamic-insert and dynamic-update attributes in a class mapping?**

**Ans:**Criteria is a simplified API for retrieving entities by composing Criterion objects. This is a very convenient approach for functionality like “search” screens where there is a variable number of conditions to be placed upon the result set.

dynamic-update (defaults to false): Specifies that UPDATE SQL should be generated at runtime and contain only those columns whose values have changed

dynamic-insert (defaults to false): Specifies that INSERT SQL should be generated at runtime and contain only the columns whose values are not null.

**What’s the use of version property in hibernate?**

**Ans:** Version property is used in hibernate to know whether an object is in transient state or in detached state.

|  |
| --- |
| Qns-3: What is the shared cache mode in second level cache? |
| Ans: To configure shared cache mode, hibernate provides javax.persistence.sharedCache.mode and it has three values.  a. ENABLE\_SELECTIVE : Entities will not be cached until entity will be annotated by cacheable.  b. DISABLE\_SELECTIVE : Those entities are cached which are explicitly not annotated with cacheable.  c. ALL : Every entities will be cached.  d. NONE: No entity will be cached. |
|  |
| What the values of CacheMode in second level cache are in hibernate? |
| Ans: There are four values of CacheMode.  a. CacheMode.NORMAL : This mode allows to read and write data in second level cache.  b. CacheMode.GET : In this mode, data will be read but will not be written in second level cache.  c. CacheMode.PUT : Data will be written but not will be read from second level cache.  d. CacheMode.REFRESH : Data will be written and will not be read from second level cache. The difference between Put and Refresh is that CacheMode.REFRESH will bypass hibernate.cache.use\_minimal\_puts effect. |
|  |
| Qns-6: What is query cache and how to configure it? |
| Ans: Hibernate can cache query results. When a query is fetched frequently then caching query result is useful. Hibernate has an overhead to enable query cache because to keep updated the query result, hibernate has to track the changes in database. We configure it as below.  hibernate.cache.use\_query\_cache = "true" |

**15**.**What the four ORM levels are in hibernate?**  
Following are the four ORM levels in hibernate:  
Pure relational (stored procedure.)

Light objects mapping (JDBC)

Medium object mapping

Full object Mapping (composition,inheritance, polymorphism, persistence by reachability)

How to implement Joins in Hibernate?

There are various ways to implement joins in hibernate.

Using associations such as one-to-one, one-to-many etc.

Using JOIN in the HQL query. There is another form “join fetch” to load associated data simultaneously, no lazy loading.

We can fire native sql query and use join key

**What are different ways to disable hibernate second level cache?**

Hibernate second level cache can be disabled using any of the following ways:

By setting use\_second\_level\_cache as false.

By using CACHEMODE.IGNORE

Using cache provider as org.hibernate.cache.NoCacheProvider

**What is Hibernate Validator Framework?**

Hibernate validator framework is the implementation of JSR303 and JSR349 provides specification for validating a bean by using annotations. Hibernate Validator provides the reference implementation of both these bean validation specification.

Can a single Hibernate Session object be used across multiple threads ?  
No, Hibernate Session is basically single-threaded and not to be used across  
multiple threads.

**How to implement Optimistic locking in Database?**

You can implement optimistic locks in your DB table in this way (This is how optimistic locking is done in Hibernate):

– Add integer “version” column to your table.

– Increase value of this column with each update of corresponding row.

– To obtain lock, just read “version” value of row.

– Add “version = obtained\_version” condition to where clause of your update statement.

– Verify number of affected rows after update. If no rows were affected – someone has already modified your entry.

Your update should look like

*UPDATE mytable SET name = ‘Andy’, version = 3 WHERE id = 1 and version = 2;*

**7. Difference between @NotNull, @NotEmpty and @NotBlank.**

**@NotNull**

CharSequence, Collection, Map or Array object cannot be null, however can be empty.

**@NotEmpty**

The CharSequence, Collection, Map or Array object cannot be null and not empty (size > 0).

**@NotBlank**

The string is not null and the length is greater than zero

8. @UniqueConstraint and @Column Unique attribute.

@UniqueConstraint and unique attribute of @Column instructs schema/DDL generation tool to generate the corresponding unique constraints however using that attributes on POJO doesn't implement constraints itself.

@Entity

@Table(uniqueConstraints=

@UniqueConstraint(columnNames = {"email", "empl\_id"}))

public class EntityClass {

...

}

@Entity

@Table(name="USER\_TABLE")

public class User{

@Id

@Column(name = "ID")

private String id;

@Column(name = "SSN", unique=true)

private String ssnumber;

**Difference between using a @OneToMany and @ElementCollection annotation in Hibernate.**

@ElementCollection maps non-entities (embeddable or basic) while @OneToMany is used to map entities.

 Name few Restriction Methods ?  
Ans. eq, ge, gt , between, in , isNull, isEmpty, isNotnull, ne , like, lt , or , not

**"What is the difference between these 2 annotations ?**

@Entity ( name ="EMPLOYEES")

@Entity

@Table ( name=""EMPLOYEES"" )

@Entity ( name="EMP")

@Table ( name="EMPLPYEES" )

Ans. First Annotation will set the Entity name as EMPLOYEES and hence will try to map with the same Table name.

The second annotation will make the Entity mapped to table EMPLOYEES irrespective of the Entity Name ( which is class name in this case ).

Third Annotations will set the different names for Enitity and Table and will explicitly map them.

What are the ways to avoid LazyInitializationException ?

Ans. 1. Set lazy=false in the hibernate config file.

2. Set @Basic(fetch=FetchType.EAGER) at the mapping.

3. Make sure that we are accessing the dependent objects before closing the session.

4. Force initialization using Hibernate.initialize

5. Using Fetch Join in HQL.

Which class elements are not persisted ?

Ans. Static and Transient.

# HIBERNATE QUERY LANGUAGE

Hibernate query language (HQL) is an object oriented query language. HQL is similar to SQL and it is case-insensitive except from the java classes and properties. Keywords like SELECT, FROM and WHERE etc. are not case sensitive but properties like table and column names are case sensitive in HQL. For understanding the HQL syntax, The HQL supports the following elements. They are

**Clauses**

**Aggregate functions**

**Subqueries**

**Association and joins**

**Expressions**

**Pagination using query**

**Clauses**

HQL support nine different clauses. They are, a)      from clause From clause is the most simplest hibernate query. This clause is used to load a complete persistent object into memory. Syntax:-

|  |  |
| --- | --- |
| [?](http://www.j2eebrain.com/java-J2ee-hibernate-query-language.html)1  2  3 | String hql = "FROM Person";  Query query = session.createQuery(hql);  List results = query.list(); |

b)      As clause As clause allows assign aliases to the classes. Synatax: –[?](http://www.j2eebrain.com/java-J2ee-hibernate-query-language.html)

|  |  |
| --- | --- |
| 1  2  3 | String hql = "FROM Employee AS E";  Query query = session.createQuery(hql);  List results = query.list(); |

The keyword ‘As’ is optional and multiple classes can appear resulting in a Cartesian product or “cross” join. Syntax1: –

|  |  |
| --- | --- |
| 1  2  3 | String hql = "FROM Person AS P";  Query query = session.createQuery(hql);  List results = query.list(); |
| [?](http://www.j2eebrain.com/java-J2ee-hibernate-query-language.html)1  2  3 | String hql = "FROM Person P, FROM Works W ";  Query query = session.createQuery(hql);  List results = query.list(); |

c)      select clause The select clause is used to obtain few properties of objects instead of the complete object. Syntax: –

|  |  |
| --- | --- |
| 1  2  3 | String hql = "SELECT P.firstName FROM  Person P";  Query query = session.createQuery(hql);  List results = query.list(); |

d)     where clause The where clause is helps to pick up the specific objects that are returned from storage. Syntax: –

|  |  |
| --- | --- |
| 1  2  3 | String hql = "FROM Person P WHERE P.id = 10";  Query query = session.createQuery(hql);  List results = query.list(); |

e)      Order by clause This clause is helps to sort our HQL query’s results. We can arrange the result by any property on the objects in the result set either ascending (ASC) or descending (DESC) order. Syntax:

|  |  |
| --- | --- |
| 1  2  3 | String hql = "FROM Person P WHERE P.id > 10 ORDER BY P.salary DESC";  Query query = session.createQuery(hql);  List results = query.list(); |

f)       Group by clause Group by clause is used to group the information based on a value of an attribute and also use the result to include an aggregate value. Syntax: –

|  |  |
| --- | --- |
| 1  2  3  4 | String hql = "SELECT SUM(P.salary), P.firtName FROM Person P” +  "GROUP BY P.firstName";  Query query = session.createQuery(hql);  List results = query.list(); |

g)      Update clause The UPDATE clause can be used to update one or more properties of one or more objects. Syntax: –

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | String hql = "UPDATE Person set salary =: salary “+  "WHERE id =:person\_id";  Query query = session.createQuery(hql);  query.setParameter("salary", 1000);  query.setParameter("person\_id", 10);  int result = query.executeUpdate();  System.out.println("Rows affected: " + result); |

h)      Delete clause The DELETE clause can be used to delete one or more objects. Syntax: –

|  |  |
| --- | --- |
| [?](http://www.j2eebrain.com/java-J2ee-hibernate-query-language.html)1  2  3  4  5  6 | String hql = "DELETE FROM Person “+  "WHERE id = :person\_id";  Query query = session.createQuery(hql);  query.setParameter("person\_id", 10);  int result = query.executeUpdate();  System.out.println("Rows affected: " + result); |

i)        Insert clause Insert into clause of hibernate query language support where records can be inserted from one object to another object. Syntax: –

|  |  |
| --- | --- |
| [?](http://www.j2eebrain.com/java-J2ee-hibernate-query-language.html)1  2  3  4  5 | String hql = "INSERT INTO Person(firstName, lastName, salary)"  +  "SELECT firstName, lastName, salary FROM old\_person";  Query query = session.createQuery(hql);  int result = query.executeUpdate();  System.out.println("Rows affected: " + result); |

**Aggregate functions**

HQL support the following five aggregate methods.

avg(property name):- This function helps to take the average of a property’s value.

count(property name or \*):- count function helps to evaluate the number times a property occurs in the results.

max(property name):- It takes the maximum value of the property values.

min(property name):- It takes the minimum value of the property values.

sum(property name):- It takes the sum total of the property values.

The distinct keyword helps to count the unique values in the row set. Syntax: –

|  |  |
| --- | --- |
| 1  2  3 | String hql = "SELECT count(distinct P.firstName) FROM Person P";  Query query = session.createQuery(hql);  List results = query.list(); |

**Subqueries**

Subqueries are query within another query. A subquery must be surrounded by parentheses (often by an SQL aggregate function call). Even correlated subqueries (subqueries that refer to an alias in the outer query) are allowed. Eg: –[?](http://www.j2eebrain.com/java-J2ee-hibernate-query-language.html)

|  |  |
| --- | --- |
| 1  2  3  4 | from Cat as fatcat  where fatcat.weight > (  select avg(cat.weight) from DomesticCat cat  ) |

**Association and joins**

We may also assign aliases to associated entities, or even to elements of a collection of values, using a join. The supported join types are borrowed from ANSI SQL

inner join left outer join right outer join

full join (not usually useful)

The inner join, left outer join and right outer join constructs may be abbreviated. Eg: –

|  |  |
| --- | --- |
| [?](http://www.j2eebrain.com/java-J2ee-hibernate-query-language.html)1  2  3 | from Cat as cat  join cat.mate as mate  left join cat.kittens as kitten |

Expressions

Expressions are permits in the where clause. They are,

mathematical operators +, -, \*, /

binary comparison operators =, >=, <=, <>, !=, like

logical operations and, or, not

string concatenation ||

SQL scalar functions like upper() and lower()

Parentheses ( ) indicate grouping  in, between, is null, not in, is not null, is empty, is not empty, member of and not member of.

“Simple” case, case ... when ... then ... else ... end, and “searched”  case, case when ... then ... else ... end

string concatenation ...||... or concat(...,...),current\_date(), current\_time(), and current\_timestamp(),second(...), minute(...), hour(...), day(...), month(...), and year(...)

Any function or operator defined by EJB-QL 3.0: substring(), trim(),  lower(),upper(), length(), locate(), abs(), sqrt(), bit\_length(), mod() ,coalesce() and nullif()

str() for converting numeric or temporal values to a readable string

cast(... as ...), where the second argument is the name of a Hibernate type, and extract(... from ...) if ANSI cast() and extract() is supported by the underlying database

The HQL index() function, that applies to aliases of a joined indexed collection

HQL functions that take collection-valued path expressions: size(), minelement(), maxelement(), minindex(), maxindex(), along with the special elements() and indicesfunctions that can be quantified using some, all, exists, any, in.

Any database-supported SQL scalar function like sign(), trunc(), rtrim(), and sin()

JDBC IN parameters

Named parameters :name, :start\_date, : x1

SQL literals ‘foo’, 69, ‘1970-01-01 10:00:01.0’

Java public static final constants eg.Color.TABBY

Pagination using query

There are two methods of the Query interface for pagination.

Query setFirstResult(int startPosition):- This method takes an integer that represents the first row in your result set, starting with row 0.

Query setMaxResults(int maxResult):- This method tells Hibernate to retrieve a fixed number maxResults of objects.

|  |  |
| --- | --- |
| 1  2  3  4  5 | String hql = "FROM Person";  Query query = session.createQuery(hql);  query.setFirstResult(1);  query.setMaxResults(10);  List results = query.list(); |

Features of Hibernate Query Language

Full support for relational operations

Return result as object

Polymorphic queries

Easy to learn

Support for advance features like pagination, fetch join with dynamic profiling, Inner/outer/full joins, Cartesian products, Projection, Aggregation (max, avg) and grouping, Ordering, Sub queries and SQL function calls.

Database independent

# HIBERNATE CRITERION

Hibernate offers a set of APIs that map the SQL functionality to objects. This allows programmers to use an entirely object-oriented approach without falling back on a relational methodology. Keep reading to find out more.

Though HQL works with the object-oriented approach, we still need to know SQL. The only difference is the replacement of the relational methodology with the object-oriented one. In essence, a developer still has to drop into an SQL-like syntax for CRUD operations.

To overcome this obstacle and provide an extensible as well as completely object-oriented solution, Hibernate exposes a set of APIs that map the SQL functionality to objects, eliminating the need for an SQL-like syntax. These APIs constitute the Hibernate Criteria Query APIs.

**Core Classes of Criteria Queries**

Each class of the Criteria Query API represents an aspect of the relational approach. There are five core APIs that are commonly used. They are:

Criteria

Criterion

Restrictions

Projection

Order

The Criteria class provides the gateway to working with criteria APIs. It is an interface that provides a simplified API for retrieving objects containing or composed of Criterion objects. In a situation where the restrictions query is composed of a variable number of fields, this approach is very useful. To get a reference to the Criteria class, use the createCriteria() method of the Session class. This method takes the name of the ORM class on which the query has to be executed. In essence the Session acts as a factory for the Criteria class. In statement form it would be:

**Criteria criteria= session.createCriteria(Order.class)**

The above statement returns a reference to Criteria for the Order ORM class.

Looking at the second core API, in the relational approach, conditions placed on retrieving [data](http://www.devarticles.com/##) are known as criterion. The Criterion class is the object-oriented representation of the relational criterion. It can be used as restrictions on the criteria query. In other words, Criterion is the object-oriented representation of the “where” clause of a SQL query. The conditions to be applied (also known as restrictions) can be provided by the Restrictions class. In code this would be:

**Criterion crit=Restriction.eq(“orderId”,”OD00009”);**  
**criteria.add(crit);**

From the above example it is clear that the Criterion is the “where” clause, which, when added to the Criteria object, provides a complete query with restrictions.Here the built-in Restriction type eq() (for testing equality) is being used.

The Restriction API provides the built-in types for Criterion. Essentially, the Restriction class is a factory to the Criterion class. All of its methods are static. In Hibernate 2.x, the Expression class provided the services that are now provided by the Restriction class. The Restriction class provides almost all the required restrictions such as equals (eq()), logical and (and()), like (like()) and so on.

The Projection class is an object-oriented representation of query resultset projection in a Criteria query. In simpler terms, projection refers to the fields mentioned in the select clause of a query. The same can be achieved by using the Projection class in a Criteria query. The Projection class acts as a factory for the Projection class. Projection can be added by using the addProjection() method of the ProjectionList class. The addProjection() method of the Criteria class in turn returns a Criterion object. In code this would be:

**List orders = session.createCriteria(Order.class)**  
**.setProjection( Projections.projectionList()**  
**.add( Projections.rowCount() )**  
**).list();**

The Order class represents the “order by” clause of SQL. By using the asc() and desc() methods of this class, order can be imposed upon the Criteria resultselt.

**Criteria Queries**

DML plays the most important role in the R (retrieve) operation of the CRUD quad. This is reflected in the core classes of Criteria API. The retrieval of [data](http://www.devarticles.com/##) itself can be separated into four major categories:

1. Projection

2. Restriction

3. Aggregation

4. Grouping

The usage of core classes among these categories cannot be generalized. The reason for this will be evident from the details. All the examples are based on the ORDER and PRODUCTS table.

The “Select” clause is just a part of the services provided by the Projection class. Following is SQL query for projection of all fields of the ORDER table in SQL:

**SELECT \* FROM ORDER**

The Criteria equivalent would be:

**List orders= session.createCriteria(Order.class).list();**

The above statement executes the corresponding SQL statement at the [database](http://www.devarticles.com/##) server, populates the instances of the Order ORM class, adds them to a list and returns the List object. Actually, the above statement is composed of two statements:

**Criteria criteria= session.createCriteria(Order.class)**and

**List orders=criteria.list()**.

The combination of such dependent statements is known as method chaining. The above code retrieves all the rows from the ORDER table. But what if only the data contained in one of the fields has to be retrieved, as in the following SQL query:

**SELECT NAME FROM PRODUCT**

Here, the Projection class comes into play. The above query can be rewritten into a Criteria query as:

**List products=session.createCriteria(Product.class)**  
**. setProjection(Projection.property(\”name\”))**  
**.list();**

It is clear from the above example that to query based on just one field, the fieldname is passed as an argument to the property() method of the Projection class. The Projection instance returned in turn becomes an argument to the setProjection() method. Similarly, to retrieve data based on two fields, ProjectionList has to be used. Hence the SQL query:

**SELECT NAME, ID FROM PRODUCT**

Would become

**List products =session.createCriteria(Product.class).setProjection(**  
**Projections.propertyList()**  
**.add(Projection.property(\”name\”))**  
**.add(Projection.property(\”id\”))**  
**)**  
**.list();**

Now let’s make the query more complex by introducing joins. What would be the equivalent of a query such as the one below:

**SELECT O.\*, P.\* FROM ORDERS O, PRODUCT P WHERE O.ORDER\_ID=P.ORDER\_ID;**

If you think the Criteria representation of the above would be as complex, then have a look at the following:

**List orders = session.createCriteria(Order.class)**  
**.setFetchMode(“products”,FetchMode.JOIN)**  
**.list();**

It’s as simple as that. The only thing to be done is to call the setFetchMode() of the Criteria class with two parameters: the name of the class with which the current class has to be joined and mode of the fetching of the data from the associated class. In the above case, the class name is actually the instance variable provided within the Order class. The mode is Join.

So retrieval is done, but there is just one problem. If the data has to be retrieved based on a condition, then what? Then Restriction has to be used.

In layman’s terms, restriction means imposing conditions. To retrieve data based on certain conditions, Restriction must be used. Here the Restriction class comes into the picture. All the conditions provided by SQL are available in Criteria. The ones most commonly used are as follows:

Restriction.**between**is used to apply a “between” constraint to the field.

Restriction.**eq**is used to apply an “equal” constraint to the field.

Restriction.**ge**is used to apply a “greater than or equal” constraint to the field.

Restriction.**gt**is used to apply a “greater than” constraint to the field.

Restriction.**idEq**is used to apply an “equal” constraint to the identifier property.

Restriction.**in**is used to apply an “in” constraint to the field.

Restriction.**isNotNull**is used to apply an “is not null” constraint to the field.

Restriction.**isNull**is used to apply an “is null” constraint to the field.

Restriction.**ne**is used to apply a “not equal” constraint to the field.

So a SQL such as this

**SELECT \* FROM ORDERS WHERE ORDER\_ID=’1092’;**

Would become

**List orders= session.createCriteria(Order.class)**  
**.add(Restrictions.eq(“orderId”,”1092”))**  
**.list();**

Applying the restrictions becomes easy in the case of joins as well. For example, the following query

**SELECT O.\*, P.\* FROM ORDERS O, PRODUCT P WHERE**

**O.ORDER\_ID=P.ORDER\_ID AND P.ID=’1111’;**

Would become

**List orders = session.createCriteria(Order.class)**  
**.setFetchMode(“products”,FetchMode.JOIN)**  
**.add(Restrictions.eq(“id”,”1111”))**  
**.list();**

Just adding the Restriction to Criteria returned by setFetchMode() does the same thing that the above given SQL does.

Through restriction conditions can be imposed on data retrieval, there are situations where the data to be retrieved has to be based on the groups of values of a column. In such conditions, Aggregation must be used. Criteria provides aggregation functionality through the Projection class itself. So to get the count of all the rows present in the ORDER table based on the ID field, the criteria query would be:

**List orders = session.createCriteria(Order.class)**  
**.setProjection( Projections.projectionList()**  
**.add( Projections.count(“id”) ))**  
**.list();**

Similarly all the aggregate functions can be used as they are provided as static functions. As shown in the above example, each function takes the field name as the argument.

When the aggregation functions are used, the values may have to be grouped according to a particular field. Grouping always operates on a dataset. In Criteria Query API, grouping is provided by the Projection class. The groupProperty() method of the Projections class provides the grouping functionality. So a query like the one given below:

**SELECT COUNT(ID) FROM ORDER HAVING PRICETOTAL>2000 GROUP BY ID**

Can be rewritten in Criteria query as follows:

**List orders = session.createCriteria(Order.class)**  
**.setProjection( Projections.projectionList()**  
**.add( Projections.count(“id”) )**  
**.add( Projections.groupProperty(“id”) )**  
**)**  
**.list();**

**A Criteria Query in the Real World**

package com.someorg.persist.op;

public class OrderOP {

SessionFactory sf;

public OrderOP(){  
Configuration cfg = new Configuration()  
.addClass(Order.class);

sf = cfg.buildSessionFactory();}

**public Order getOrder(String lower, String upper){**  
**// open session**  
**Session sess = sf.openSession();**

//The following code has been commented so that  
//comparison between HQL and Criteria Query

/\*String query = “select o from o ”  
+ “Order as o join o.products as p ”  
+ “where o.priceTotal > :priceTotalLower ”  
+ “and o.priceTotal < :priceTotalUpper”;

Query q = sess.createQuery(query);  
q.setDouble(“priceTotalLower”,  
Double.parseDouble(lower));

q.setDouble(“priceTotalUpper”,  
Double.parseDouble(upper));  
**\*/**  
**List list = sess.createCriteria(Order.class)**  
**.add(Restrictions.between(lower,upper)**  
**. list();**

**Order o=(Order)list.iterator.next();**

**return o;}**

public static void main(String args[]){  
Order o=OrderOP().getOrder(“2000’,”3000”);

System.out.println(“Order Id:”+ o.id);  
//and so on

# Hibernate JPA Annotations - Contents:

|  |  |
| --- | --- |
| Annotation | Package Detail/Import statement |
| [@Entity](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Entity) | import javax.persistence.Entity; |
| [@Table](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Table) | import javax.persistence.Table; |
| [@Column](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Column) | import javax.persistence.Column; |
| [@Id](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Id) | import javax.persistence.Id; |
| [@GeneratedValue](http://www.techferry.com/articles/hibernate-jpa-annotations.html#GeneratedValue) | import javax.persistence.GeneratedValue; |
| [@Version](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Version) | import javax.persistence.Version; |
| [@OrderBy](http://www.techferry.com/articles/hibernate-jpa-annotations.html#OrderBy) | import javax.persistence.OrderBy; |
| [@Transient](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Transient) | import javax.persistence.Transient; |
| [@Lob](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Lob) | import javax.persistence.Lob; |
| [Hibernate Association Mapping Annotations](http://www.techferry.com/articles/hibernate-jpa-annotations.html#HibernateAssociations) | |
| [@OneToOne](http://www.techferry.com/articles/hibernate-jpa-annotations.html#OneToOne) | import javax.persistence.OneToOne; |
| [@ManyToOne](http://www.techferry.com/articles/hibernate-jpa-annotations.html#ManyToOne) | import javax.persistence.ManyToOne; |
| [@OneToMany](http://www.techferry.com/articles/hibernate-jpa-annotations.html#OneToMany) | import javax.persistence.OneToMany; |
| [@ManyToMany](http://www.techferry.com/articles/hibernate-jpa-annotations.html#ManyToMany) | import javax.persistence.ManyToMany; |
| [@PrimaryKeyJoinColumn](http://www.techferry.com/articles/hibernate-jpa-annotations.html#PrimaryKeyJoinColumn) | import javax.persistence.PrimaryKeyJoinColumn; |
| [@JoinColumn](http://www.techferry.com/articles/hibernate-jpa-annotations.html#JoinColumn) | import javax.persistence.JoinColumn; |
| [@JoinTable](http://www.techferry.com/articles/hibernate-jpa-annotations.html#JoinTable) | import javax.persistence.JoinTable; |
| [@MapsId](http://www.techferry.com/articles/hibernate-jpa-annotations.html#MapsId) | import javax.persistence.MapsId; |
| [Hibernate Inheritance Mapping Annotations](http://www.techferry.com/articles/hibernate-jpa-annotations.html#HibernateInheritanceMapping) | |
| [@Inheritance](http://www.techferry.com/articles/hibernate-jpa-annotations.html#Inheritance) | import javax.persistence.Inheritance; |
| [@DiscriminatorColumn](http://www.techferry.com/articles/hibernate-jpa-annotations.html#DiscriminatorColumn) | import javax.persistence.DiscriminatorColumn; |
| [@DiscriminatorValue](http://www.techferry.com/articles/hibernate-jpa-annotations.html#DiscriminatorValue) | import javax.persistence.DiscriminatorValue; |

@Entity

Annotate all your entity beans with @Entity.

|  |  |
| --- | --- |
| 1  2  3  4 | @Entity  public class Company implements Serializable {  ...} |

@Table

Specify the database table this Entity maps to using the name attribute of @Table annotation. In the example below, the data will be stored in 'company' table in the database.

|  |  |
| --- | --- |
| 1  2  3  4  5 | @Entity  @Table(name = "company")  public class Company implements Serializable {  ...  } |

@Column

Specify the column mapping using @Column annotation.

|  |  |
| --- | --- |
| 13  4  5  6  7  8  9 | @Entity  @Table(name = "company")  public class Company implements Serializable {     @Column(name = "name")    private String name;  ...  } |

@Id

Annotate the id column using @Id.

|  |  |
| --- | --- |
| 14  5  6  7  8  9  10 | @Entity  @Table(name = "company")  public class Company implements Serializable {    @Id    @Column(name = "id")    private int id;  ...} |

@GeneratedValue

Let database generate (auto-increment) the id column.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Entity  @Table(name = "company")  public class Company implements Serializable {    @Id    @Column(name = "id")    @GeneratedValue    private int id;  ...} |

@Version

Control versioning or concurrency using @Version annotation.

|  |  |
| --- | --- |
| 15  6  7  8  9  10 | @Entity  @Table(name = "company")  public class Company implements Serializable {    @Version    @Column(name = "version")    private Date version;  ...} |

@OrderBy

Sort your data using @OrderBy annotation. In example below, it will sort all contacts in a company by their firstname in ascending order.

|  |  |
| --- | --- |
| 1  2 | @OrderBy("firstName asc")  private Set contacts; |

@Transient

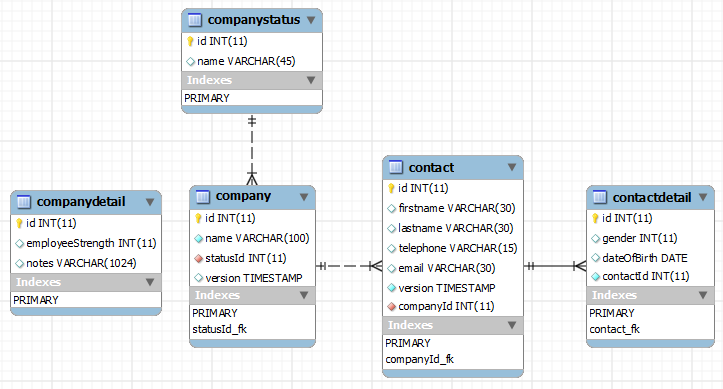
Annotate your transient properties with @Transient.

@Lob

Annotate large objects with @Lob.

Hibernate Association Mapping Annotations

Example App DB Schema

   
The database for this tutorial is designed to illustrate various association mapping concepts.   
In RDBMS implementations, entities are joined using the following ways:

Shared Primary Key

Foreign Key

Association Table

In our example app,

Tables company and companyDetail have shared values for primary key. It is a one-to-one assoication.

Tables contact and contactDetail are linked through a foreign key. It is also a one to one association.

Tables contact and company are linked through a foriegn key in many-to-one association with contact being the owner.

Tables company and companyStatus are linked through a foreign key in many-to-one association with company being the owner.

@OneToOne

|  |  |
| --- | --- |
| Hibernate Annotation Tip | Use @PrimaryKeyJoinColumn for associated entities sharing the same primary key.  Use @JoinColumn & @OneToOne mappedBy attribute when foreign key is held by one of the entities.  Use @JoinTable and mappedBy entities linked through an association table.  Persist two entities with shared key using @MapsId |

For entities Company and CompanyDetail sharing the same primary key, we can associate them using @OneToOne and @PrimaryKeyJoinColumn as shown in the example below.   
Notice that the id property of CompanyDetail is NOT annotated with @GeneratedValue. It will be populated by id value of Company.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  1721  22  23 | @Entity  @Table(name = "company")  public class Company implements Serializable {    @Id    @Column(name = "id")    @GeneratedValue    private int id;    @OneToOne(cascade = CascadeType.MERGE)    @PrimaryKeyJoinColumn    private CompanyDetail companyDetail;    ...}   @Entity  @Table(name = "companyDetail")  public class CompanyDetail implements Serializable {     @Id    @Column(name = "id")    private int id;    ...} |

For entities Contact and ContactDetail linked through a foriegn key, we can use @OneToOne and @JoinColumn annotations. In example below, the id genereated for Contact will be mapped to 'contact\_id' column of ContactDetail table. Please note the usage of @MapsId for the same.

|  |  |
| --- | --- |
| 1  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | @Entity  @Table(name = "contactDetail")  public class ContactDetail implements Serializable {    @Id    @Column(name = "id")    @GeneratedValue    private int id;    @OneToOne    @MapsId    @JoinColumn(name = "contactId")    private Contact contact;    ...}  @Entity  @Table(name = "contact")  public class Contact implements Serializable {    @Id    @Column(name = "ID")    @GeneratedValue    private Integer id;     @OneToOne(mappedBy = "contact", cascade = CascadeType.ALL)    private ContactDetail contactDetail;    ....} |

Also note that the relationship between Company and CompanyDetail is uni-directional. On the other hand, the relationship between Contact and Contact Detail is bi-directional and that can be achieved using 'mappedBy' attribute.   
  
The rationale to have one relationship as uni-directional and other as bi-directional in this tutorial is to illustrate both concepts and their usage. You can opt for uni-directional or bi-directional relationships to suit your needs.

@ManyToOne

|  |  |
| --- | --- |
| Hibernate Annotation Tip | Use @JoinColumn when foreign key is held by one of the entities.  Use @JoinTable for entities linked through an association table. |

The two examples below illustrate many-to-one relationships. Contact to Company and Company to CompanyStatus. Many contacts can belong to a company. Similary many companies can share the same status (Lead, Prospect, Customer) - there will be many companies that are currently leads.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @Entity  @Table(name = "contact")  public class Contact implements Serializable {    @ManyToOne    @JoinColumn(name = "companyId")    private Company company;    ... }  @Entity  @Table(name = "company")  public class Company implements Serializable {    @ManyToOne    @JoinColumn(name = "statusId")    private CompanyStatus status;    ...    } |

@OneToMany

|  |  |
| --- | --- |
| Hibernate Annotation Tip | Use mappedBy attribute for bi-directional associations with ManyToOne being the owner.  OneToMany being the owner or unidirectional with foreign key - try to avoid such associations but can be achieved with @JoinColumn  @JoinTable for Unidirectional with association table |

Please see the many-to-one relationship between Contact and Company above. Company to Contact will be a one-to-many relationship. The owner of this relationship is Contact and hence we will use 'mappedBy' attribute in Company to make it bi-directional relationship.

|  |  |
| --- | --- |
| 4  5  6  7  8  9  10  11 | @Entity  @Table(name = "company")  public class Company implements Serializable {    @OneToMany(mappedBy = "company", fetch = FetchType.EAGER)    @OrderBy("firstName asc")    private Set contacts;    ...   } |

Again, for this tutorial, we have kept Company to CompanyStatus relationship as uni-directional.

@ManyToMany

|  |  |
| --- | --- |
| Hibernate Annotation Tip | Use @JoinTable for entities linked through an association table.  Use mappedBy attribute for bi-directional association. |

@PrimaryKeyJoinColumn

@PrimaryKeyJoinColumn annotation is used for associated entities sharing the same primary key. See [OneToOne](http://www.techferry.com/articles/hibernate-jpa-annotations.html#OneToOne) section for details.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @Entity  @Table(name = "company")  public class Company implements Serializable {    @Id    @Column(name = "id")    @GeneratedValue    private int id;      @OneToOne(cascade = CascadeType.MERGE)    @PrimaryKeyJoinColumn    private CompanyDetail companyDetail;     ...} |

@JoinColumn

Use @JoinColumn annotation for one-to-one or many-to-one associations when foreign key is held by one of the entities. We can use @OneToOne or @ManyToOne mappedBy attribute for bi-directional relations. Also see [OneToOne](http://www.techferry.com/articles/hibernate-jpa-annotations.html#OneToOne) and [ManyToOne](http://www.techferry.com/articles/hibernate-jpa-annotations.html#ManyToOne) sections for more details.

|  |  |
| --- | --- |
| 1  2  3 | @ManyToOne  @JoinColumn(name = "statusId")  private CompanyStatus status; |

@JoinTable

Use @JoinTable and mappedBy for entities linked through an association table.

@MapsId

Persist two entities with shared key (when one entity holds a foreign key to the other) using @MapsId annotation. See [OneToOne](http://www.techferry.com/articles/hibernate-jpa-annotations.html#OneToOne) section for details.

|  |  |
| --- | --- |
| 1  2  3  4 | @OneToOne  @MapsId  @JoinColumn(name = "contactId")  private Contact contact; |

Hibernate Inheritance Mapping Annotations

To understand Inheritance Mapping annotations, you must first understand [Inheritance Mapping in Hiberate](http://docs.jboss.org/hibernate/core/3.5/reference/en/html/inheritance.html) in detail. Once you understand Inheritance mapping concepts, please review below for annotations to be used.

table per class hierarchy - single table per Class Hierarchy Strategy: the <subclass> element in Hibernate

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Entity  @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  @DiscriminatorColumn(name="planetype", discriminatorType=DiscriminatorType.STRING )    @DiscriminatorValue("Plane")  public class Plane { ... }    @Entity  @DiscriminatorValue("A320")  public class A320 extends Plane { ... } |

table per class/subclass - joined subclass Strategy: the <joined-subclass> element in Hibernate

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Entity  @Inheritance(strategy=InheritanceType.JOINED)  public class Boat implements Serializable { ... }    @Entity  @PrimaryKeyJoinColumn  public class Ferry extends Boat { ... } |

table per concrete class - table per Class Strategy: the <union-class> element in Hibernate

|  |  |  |
| --- | --- | --- |
| 1  2  3 | @Entity  @Inheritance(strategy = InheritanceType.TABLE\_PER\_CLASS)  public class Flight implements Serializable { ... } | |
| Hibernate Annotation Tip | Note: This strategy does not support the IDENTITY generator strategy: the id has to be shared across several tables. Consequently, when using this strategy, you should not use AUTO nor IDENTITY. |

@Inheritance

See [Hibernate Inheritance Mapping Annotations](http://www.techferry.com/articles/hibernate-jpa-annotations.html#HibernateInheritanceMapping) section for details.

|  |  |
| --- | --- |
| 1  2 | @Entity  @Inheritance(strategy=InheritanceType.SINGLE\_TABLE) |

@DiscriminatorColumn

See [Hibernate Inheritance Mapping Annotations](http://www.techferry.com/articles/hibernate-jpa-annotations.html#HibernateInheritanceMapping) section for details.

|  |  |
| --- | --- |
| 1  2  3 | @Entity  @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  @DiscriminatorColumn(name="planetype", discriminatorType=DiscriminatorType.STRING ) |

@DiscriminatorValue

See [Hibernate Inheritance Mapping Annotations](http://www.techferry.com/articles/hibernate-jpa-annotations.html#HibernateInheritanceMapping) section for details.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Entity  @Inheritance(strategy=InheritanceType.SINGLE\_TABLE)  @DiscriminatorColumn(name="planetype", discriminatorType=DiscriminatorType.STRING )    @DiscriminatorValue("Plane")  public class Plane { ... }    @Entity  @DiscriminatorValue("A320")  public class A320 extends Plane { ... } |

References:

[Hibernate Annotations](http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html_single/): http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html\_single/

[Inheritance Mapping Reference](http://docs.jboss.org/hibernate/core/3.5/reference/en/html/inheritance.html): http://docs.jboss.org/hibernate/core/3.5/reference/en/html/inheritance.html

# Associations

Association mappings are one of the key features of JPA and [Hibernate](http://www.hibernate.org/). They model the relationship between two database tables as attributes in your domain model. That allows you to easily navigate the associations in your domain model and [JPQL](https://www.thoughts-on-java.org/jpql/) or Criteria queries.

You can map each of them as a uni- or bidirectional association. That means you can either model them as an attribute on only one of the associated entities or on both. That has no impact on your database mapping, but it defines in which direction you can use the relationship in your domain model and JPQL or Criteria queries. I will explain that in more details in the first example.

Many-to-One Associations

An order consists of multiple items, but each item belongs to only one order. That is a typical example for a many-to-one association. If you want to model this in your database model, you need to store the [primary key](https://www.thoughts-on-java.org/jpa-generate-primary-keys/) of the Order record as a foreign key in the OrderItem table.

With JPA and Hibernate, you can model this in 3 different ways. You can either model it as a bidirectional association with an attribute on the Order and the OrderItem entity. Or you can model it as a unidirectional relationship with an attribute on the Order or the OrderItem entity.

Unidirectional Many-to-One Association

Let’s take a look at the unidirectional mapping on the OrderItem entity first. The OrderItementity represents the many side of the relationship and the OrderItem table contains the foreign key of the record in the Order table.

As you can see in the following code snippet, you can model this association with an attribute of type Order and a @ManyToOne annotation. The Order order attribute models the association, and the annotation tells Hibernate how to map it to the database.

|  |  |
| --- | --- |
| 1 | @Entity  public class OrderItem {      @ManyToOne      private Order order;      …} |

That is all you need to do to model this association. By default, Hibernate generates the name of the foreign key column based on the name of the relationship mapping attribute and the name of the primary key attribute. In this example, Hibernate would use a column with the name order\_id to store the foreign key to the Order entity.

If you want to use a different column, you need to define the foreign key column name with a @JoinColumn annotation. The example in the following code snippet tells Hibernate to use the column fk\_order to store the foreign key.

|  |  |
| --- | --- |
|  | @Entity  public class OrderItem {      @ManyToOne      @JoinColumn(name = “fk\_order”)      private Order order;      …} |

You can now use this association in your business code to get the Order for a given OrderItemand to add or remove an OrderItem to or from an existing Order.

|  |  |
| --- | --- |
| 1 | Order o = em.find(Order.class, 1L);   OrderItem i = new OrderItem();  i.setOrder(o);  em.persist(i); |

That’s all about the mapping of unidirectional many-to-one associations for now. If you want to dive deeper, you should take a look at FetchTypes. I explained them in detail in my [Introduction to JPA FetchTypes](https://www.thoughts-on-java.org/entity-mappings-introduction-jpa-fetchtypes/).

But now, let’s continue with the association mappings and talk about unidirectional one-to-many relationships next. As you might expect, the mapping is very similar to this one.

Unidirectional One-to-Many Association

The unidirectional one-to-many relationship mapping is not very common. In the example of this post, it only models the association on the Order entity and not on the OrderItem.

The basic mapping definition is very similar to the many-to-one association. It consist of the List items attribute which stores the associated entities and a @OneToMany association.

|  |  |
| --- | --- |
|  | @Entity  public class Order {      @OneToMany      private List<OrderItem> items = new ArrayList<OrderItem>();      …} |

But this is most likely not the mapping you’re looking for because Hibernate uses an association table to map the relationship. If you want to avoid that, you need to use a @JoinColumnannotation to specify the foreign key column.

The following code snippet shows an example of such a mapping. The @JoinColumn annotation tells Hibernate to use the fk\_order column in the OrderItem table to join the two database tables.

|  |  |
| --- | --- |
| 125  6  7  8  9 | @Entity  public class Order {      @OneToMany      @JoinColumn(name = “fk\_order”)      private List<OrderItem> items = new ArrayList<OrderItem>();       …} |

You can now use the association in your business code to get all OrderItems of an Order and to add or remove an OrderItem to or from an Order.

|  |  |
| --- | --- |
| 1 | Order o = em.find(Order.class, 1L);  OrderItem i = new OrderItem();  o.getItems().add(i);  em.persist(i); |

 Bidirectional Many-to-One Associations

The bidirectional Many-to-One association mapping is the most common way to model this relationship with JPA and Hibernate. It uses an attribute on the Order and the OrderItem entity. This allows you to navigate the association in both directions in your domain model and your [JPQL queries](https://www.thoughts-on-java.org/jpql/).

The mapping definition consists of 2 parts:

the to-many side of the association which owns the relationship mapping and

the to-one side which just references the mapping

Let’s take a look at the owning side first. You already know this mapping from the [unidirectional Many-to-One](https://www.thoughts-on-java.org/ultimate-guide-association-mappings-jpa-hibernate/#uniManyToOne) association mapping. It consists of the Order order attribute, a @ManyToOneannotation and an optional @JoinColumn annotation.

The owning part of the association mapping already provides all the information Hibernate needs to map it to the database. That makes the definition of the referencing part simple. You just need to reference the owning association mapping. You can do that by providing the name of the association-mapping attribute to the mappedBy attribute of the @OneToMany annotation. In this example, that’s the order attribute of the OrderItem entity.

|  |  |
| --- | --- |
| 13  5  6  7  8 | @Entity  public class Order {      @OneToMany(mappedBy = “order”)      private List<OrderItem> items = new ArrayList<OrderItem>();     …} |

You can now use this association in a similar way as the unidirectional relationships I showed you before. But adding and removing an entity from the relationship requires an additional step. You need to update both sides of the association.

|  |  |
| --- | --- |
| 1  5  6  7  8 | Order o = em.find(Order.class, 1L);  OrderItem i = new OrderItem();  i.setOrder(o);  o.getItems().add(i);  em.persist(i); |

That is an error-prone task, and a lot of developers prefer to implement it in a utility method which updates both entities.

|  |  |
| --- | --- |
| 1  7  8  9  10 | @Entity  public class Order {    …               public void addItem(OrderItem item) {          this.items.add(item);          item.setOrder(this);      }    …  } |

That’s all about many-to-one association mapping for now. You should also take a look at FetchTypes and how they impact the way Hibernate loads entities from the database. I get into detail about them in my [Introduction to JPA FetchTypes](https://www.thoughts-on-java.org/entity-mappings-introduction-jpa-fetchtypes/).

Many-to-Many Associations

Many-to-Many relationships are another often used association type. On the database level, it requires an additional association table which contains the [primary key](https://www.thoughts-on-java.org/jpa-generate-primary-keys/) pairs of the associated entities. But as you will see, you don’t need to map this table to an entity.

A typical example for such a many-to-many association are Products and Stores. Each Storesells multiple Products and each Product gets sold in multiple Stores.

Similar to the many-to-one association, you can model a many-to-many relationship as a uni- or bidirectional relationship between two entities.

But there is an important difference that might not be obvious when you look at the following code snippets. When you map a many-to-many association, you should use a Set instead of a List as the attribute type. Otherwise, Hibernate will take a very inefficient approach to remove entities from the association. It will remove all records from the association table and re-insert the remaining ones. You can avoid that by using a Set instead of a List as the attribute type.

OK let’s take a look at the unidirectional mapping first.

Unidirectional Many-to-Many Associations

Similar to the previously discussed mappings, the unidirectional many-to-many relationship mapping requires an entity attribute and a @ManyToMany annotation. The attribute models the association and you can use it to navigate it in your domain model or [JPQL queries](https://www.thoughts-on-java.org/jpql/). The annotation tells Hibernate to map a many-to-many association.

Let’s take a look at the relationship mapping between a Store and a Product. The Set productsattribute models the association in the domain model and the @ManyToMany association tells Hibernate to map it as a many-to-many association.

And as I already explained, please note the difference to the previous many-to-one mappings. You should map the associated entities to a Set instead of a List.

|  |  |
| --- | --- |
| 14  5  6  7  8 | @Entity  public class Store {      @ManyToMany      private Set<Store> products = new HashSet<Store>();       …} |

If you don’t provide any additional information, Hibernate uses its default mapping which expects an association table with the name of both entities and the primary key attributes of both entities. In this case, Hibernate uses the Store\_Product table with the columns store\_idand product\_id.

You can customize that with a @JoinTable annotation and its attributes joinColumns and inverseJoinColumns. The joinColumns attribute defines the foreign key columns for the entity on which you define the association mapping. The inverseJoinColumns attribute specifies the foreign key columns of the associated entity.

The following code snippet shows a mapping that tells Hibernate to use the store\_product table with the fk\_product column as the foreign key to the Product table and the fk\_store column as the foreign key to the Store table.

|  |  |
| --- | --- |
|  | @Entity  public class Store {      @ManyToMany      @JoinTable(name = “store\_product”,             joinColumns = { @JoinColumn(name = “fk\_store”) },             inverseJoinColumns = { @JoinColumn(name = “fk\_product”) })      private Set<Product> products = new HashSet<Product>();      …} |

That’s all you have to do to define an unidirectional many-to-many association between two entities. You can now use it to get a Set of associated entities in your domain model or to join the mapped tables in a JPQL query.

|  |  |
| --- | --- |
| 1  5  6  7 | Store s = em.find(Store.class, 1L);   Product p = new Product();  s.getProducts().add(p);   em.persist(p); |

Bidirectional Many-to-Many Associations

The bidirectional relationship mapping allows you to navigate the association in both directions. And after you’ve read the post this far, you’re probably not surprised when I tell you that the mapping follows the same concept as the bidirectional mapping of a many-to-one relationship.

One of the two entities owns the association and provides all mapping information. The other entity just references the association mapping so that Hibernate knows where it can get the required information.

Let’s start with the entity that owns the relationship. The mapping is identical to the [unidirectional many-to-many association](https://www.thoughts-on-java.org/ultimate-guide-association-mappings-jpa-hibernate/#uniManyToMany) mapping. You need an attribute that maps the association in your domain model and a @ManyToMany association. If you want to adapt the default mapping, you can do that with a @JoinColumn annotation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | @Entity  public class Store {      @ManyToMany      @JoinTable(name = “store\_product”,             joinColumns = { @JoinColumn(name = “fk\_store”) },             inverseJoinColumns = { @JoinColumn(name = “fk\_product”) })      private Set<Product> products = new HashSet<Product>();      …} |

The mapping for the referencing side of the relationship is a lot easier. Similar to the [bidirectional many-to-one relationship mapping](https://www.thoughts-on-java.org/ultimate-guide-association-mappings-jpa-hibernate/biManyToOne), you just need to reference the attribute that owns the association.

You can see an example of such a mapping in the following code snippet. The List productsattribute of the Store entity owns the association. So, you only need to provide the String“products” to the mappedBy attribute of the @ManyToMany annotation.

|  |  |
| --- | --- |
|  | @Entity  public class Product{      @ManyToMany(mappedBy=”products”)      private Set<Store> stores = new HashSet<Store>();      …} |

That’s all you need to do to define a bidirectional many-to-many association between two entities. But there is another thing you should do to make it easier to use the bidirectional relationship.

You need to update both ends of a bidirectional association when you want to add or remove an entity. Doing that in your business code is verbose and error-prone. It’s, therefore, a good practice to provide helper methods which update the associated entities.

|  |  |
| --- | --- |
| 15  6  7  8  9  10  11  12  13  14  15 | @Entity  public class Store {      public void addProduct(Product p) {          this.products.add(p);          p.getStores().add(this);      }      public void removeProduct(Product p) {          this.products.remove(p);          p.getStores().remove(this);      }  } |

OK, now we’re done with the definition of the many-to-many association mappings. Let’s take a look at the third and final kind of association: The one-to-one relationship.

One-to-One Associations

One-to-one relationships are rarely used in relational table models. You, therefore, won’t need this mapping too often. But you will run into it from time to time. So it’s good to know that you can map it in a similar way as all the other associations.

An example for a one-to-one association could be a Customer and the ShippingAddress. Each Customer has exactly one ShippingAddress and each ShippingAddress belongs to one Customer. On the database level, this mapped by a foreign key column either on the ShippingAddress or the Customer table.

Let’s take a look at the unidirectional mapping first.

Unidirectional One-to-One Associations

As in the previous unidirectional mapping, you only need to model it for the entity for which you want to navigate the relationship in your query or domain model. Let’s say you want to get from the Customer to the ShippingAddress entity.

The required mapping is similar to the previously discussed mappings. You need an entity attribute that represents the association, and you have to annotate it with an @OneToOneannotation.

When you do that, Hibernate uses the name of the associated entity and the name of its primary key attribute to generate the name of the foreign key column. In this example, it would use the column shippingaddress\_id. You can customize the name of the foreign key column with a @JoinColumn annotation. The following code snippet shows an example of such a mapping.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Entity  public class Customer{      @OneToOne      @JoinColumn(name = “fk\_shippingaddress”)      private ShippingAddress shippingAddress;       …  } |

That’s all you need to do to define a one-to-one association mapping. You can now use it in your business to add or remove an association, to navigate it in your domain model or to join it in a JPQL query.

|  |  |
| --- | --- |
| 1  2 | Customer c = em.find(Customer.class, 1L);  ShippingAddress sa = c.getShippingAddress(); |

Bidirectional One-to-One Associations

The bidirectional one-to-one relationship mapping extends the unidirectional mapping so that you can also navigate it in the other direction. In this example, you also model it on the ShippingAddress entity so that you can get the Customer for a giving ShippingAddress.

Similar to the previously discussed bidirectional mappings, the bidirectional one-to-one relationship consists of an owning and a referencing side. The owning side of the association defines the mapping, and the referencing one just links to that mapping.

The definition of the owning side of the mapping is identical to the unidirectional mapping. It consists of an attribute that models the relationship and is annotated with a @OneToOneannotation and an optional @JoinColumn annotation.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | @Entity  public class Customer{      @OneToOne      @JoinColumn(name = “fk\_shippingaddress”)      private ShippingAddress shippingAddress;       …  } |

The referencing side of the association just links to the attribute that owns the relationship. Hibernate gets all information from the referenced mapping, and you don’t need to provide any additional information. You can define that with the mappedBy attribute of the @OneToOneannotation. The following code snippet shows an example of such a mapping.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Entity  public class ShippingAddress{        @OneToOne(mappedBy = “shippingAddress”)      private Customer customer;        …  } |

Summary

The relational table model uses many-to-many, many-to-one and one-to-one associations to model the relationship between database records. You can map the same relationships with JPA and Hibernate, and you can do that in an unidirectional or bidirectional way.

The unidirectional mapping defines the relationship only on 1 of the 2 associated entities, and you can only navigate it in that direction. The bidirectional mapping models the relationship for both entities so that you can navigate it in both directions.

The concept for the mapping of all 3 kinds of relationships is the same.

If you want to create an unidirectional mapping, you need an entity attribute that models the association and that is annotated with a @ManyToMany, @ManyToOne, @OneToMany or @OneToOne annotation. Hibernate will generate the name of the required foreign key columns and tables based on the name of the entities and their primary key attributes.

The bidirectional associations consist of an owning and a referencing side. The owning side of the association is identical to the unidirectional mapping and defines the mapping. The referencing side only links to the attribute that owns the association.

# Hibernate Caching

http://www.wideskills.com/hibernate/hibernate-caching

**19.1 Overview**

Caching is a feature that is being used in applications to improve the performance. Cache is positioned between the database and the application and usually the data of database queries gets cached (local copy) in disk or memory so that subsequent calls for the same data can be served by cache only.

Hibernate supports caching at a class level, collections level and even at a query level.

**19.2 Caching Scopes**

Cache can be in any one of the below three scopes

**Transactional Scope** – The scope of this type of cache is limited to the unit of work (transaction). Each transaction has its own  copy so it will not be shared and will not be accessed concurrently.

      If the same query is fired in same session again, it will be served by cache second time but in different session, it will not be served by cache.

**Process Scope -**  In this scope, cache is shared between transactions (or we can say between sessions) and between multiple threads. If the same query is fired in different session also, it will be served by cache. As data retrieved by one session will be visible to another session as well, we have to use this scope carefully.

      Note: Either a  persistent instances itself or  persistent state in a disassembled format can be stored in process cache. Each transaction  accessing the cache  needs to reassemble at instance from  cached data.

**Cluster Scope-**This scope is shared between process running on different machines (in cluster )as well.

**19.3 Hibernate Cache Architecture**

Hibernate has a two levels of  cache.

**First Level Cache –** The scope of this transaction is at a session level and  is  enabled for session by default. If the same query is fired in same session again, it will be served by cache second time, but in a different session, it will not be served by cache.  (transactional scope)

**Second Level Cache –** Second level cache can be enabled for a process or cluster scope. In this cache the state of a persistent object is stored (in a disassembled) form and not as a complete instance. Second level cache is optional and can be configured to cache at a class, collection level.

        There are several cache providers available and supported by Hibernate which can be plugged to use.

        It is very important to know that second  level cache is at a SessionFactory level, which means all sessions of same **session factory** will share the cache data.

**19.4 Concurrency Strategies**

The main responsibility of Concurrency Strategy is to store and retrieve data from the cache. There are four in-built concurrency strategies –

**Read Only** –As its name suggests, it is appropriate for the data which is never changed like reference data (like month in a year)

Used only for entities that never change (exception is thrown if an attempt to update such an entity is made). It is very simple and performant. Very suitable for some static reference data that don’t change

**Non Strict Read Write** – High possibility that we can read stale data and inconsistency between the database and a cache. Should be used when data hardly changes.

Cache is updated after a transaction that changed the affected data has been committed. Thus, strong consistency is not guaranteed and there is a small time window in which stale data may be obtained from cache. This kind of strategy is suitable for use cases that can tolerate eventual consistency

**Read-Write-**  this strategy maintains read-committed isolation level (read a committed data) and is not available in clustered environments.

This strategy guarantees strong consistency which it achieves by using ‘soft’ locks: When a cached entity is updated, a soft lock is stored in the cache for that entity as well, which is released after the transaction is committed. All concurrent transactions that access soft-locked entries will fetch the corresponding data directly from database

**Transactional**-It prevents stale data, so should be used when stale data is not acceptable at all.

Custom concurrency strategy can be written by implementing org.hibernate.cache.CacheConcurrencyStrategy interface .

**19.5 Cache Providers**

Hibernate does provide built-in support for following four Cache Providers

**EHCache**- This is a widely used open source cache framework and can cache the data in memory or in disk.  EHCache also supports Query cache. Complete Details about EHCache can be grabbed from [http://ehcache.org](http://ehcache.org/). EHCahce does not support Transaction Concurrency Strategy.

**OSCache (Open Symphony)** – Similar to EHCahce , OSCache is also open source cache framework and supports cluster cache as well. Details can be found at [www.opensymphony.com/oscache/](http://www.opensymphony.com/oscache/)

**SwarmCache**- SwarmCache does not support Hibernate Query cache and uses IP multicasting to support distributed cache. More details about SwarmCache is available on [http://swarmcache.sourceforge.net](http://swarmcache.sourceforge.net/).

**JBoss Cache** –This is also a distributed Cache framework and supports Hibernate Query Cache. JBoss cache replicates the data across all nodes in distributed system. More details about JBoss Cache can be read at [www.jboss.org/jbosscache/](http://www.jboss.org/jbosscache/)

We need to configure the cache provider in hibernate.cfg.xml file using “hibernate.cache.provider\_class” property. Value of this property is the cache provider class name (refer Section 15.6)

**19.6 Cache Providers Support for Concurrency Strategies**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Provider** | **Provider Class** | **Read Only** | **Non Strict Read Write** | **Read Write** | **Transactional** |
| EHCache | org.hibernate.cache.EHCacheProvider | Yes | Yes | Yes |  |
| OSCache | org.hibernate.cache.OSCacheProvider | Yes | Yes | Yes |  |
| Swarm Cache | org.hibernate.cache.SwarmCacheProvider | Yes | Yes | Yes |  |
| JBoss Cache | org.hibernate.cache.JBossCacheProvider | Yes |  |  | Yes |

**19.7 Cache Region**

Cache Region can be thought of as a named area and Hibernate does cache each class, collection in a different cache region. Each Cache Region conceptually stores the persistent state (values in disassembled form) and the identifier of the containing collections or other classes.

The name of a cache region is the fully qualified name of the class and for the  containing class/collections  it collection name together with qualified class name of the parent class. For example “if Book class has a collection of chapters “ then “com.hibenrate.tutorial.Book” and “com.hibernate.tutorial.Book.chapters” will be cache region names.

If the application is working with multiple Session Factories than cache names may conflict so we can use prefix support of Hibernate.The Hibernate supports a property named hibernate.cache.region\_prefix which can be used to specify cache region name prefix of  a SessionFactory. For example, if property is set as “sf1” then Book cache region will become “sf1.com.hibenrate.tutorial.Book”

**19.8 Example – First Level Cache**

As we discussed earlier that first level cache is the default cache and is at a session level. Let’s write an example to validate our understanding.

**Student Entity**

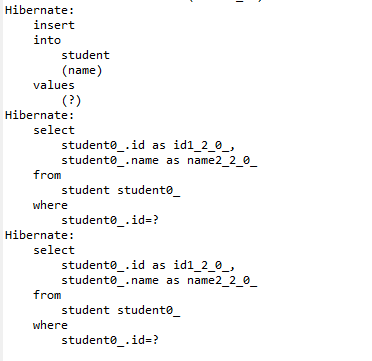
|  |  |
| --- | --- |
| [?](http://www.wideskills.com/hibernate/hibernate-caching)  2  3  4  5  6 | package com.tutorial.hibernate;   public class Student {       private int id;      private String name;  //setters and getters  } |

**Test Program**

|  |  |
| --- | --- |
|  | public class Test {       public static void main(String args[])     {          Configuration cfg = new Configuration().configure();          SessionFactory factory = cfg.buildSessionFactory();          Session session=  factory.openSession();          Transaction tx = session.beginTransaction();          Student student = new Student();          student.setName("Student A");            int id = (Integer)session.save(student);          tx.commit();                        session.close();          session=  factory.openSession();          Student st1 = (Student) session.get(Student.class, id);          Student st2 = (Student) session.get(Student.class, id);          session.close();          session=  factory.openSession();          Student st3 = (Student) session.get(Student.class, id);          session.close();          factory.close();        }} |

**Result**

On running Test Program we ca\n validate that only one select query is being fired for st1 and st2 because they are in the same session, but when again same student with same identifier is fetched in a different session, query is again fired because First level cache is session or transaction based.



**19.9 Second Level Caching Configuration**

There are certain steps need to follow to enable and configure the second level cache.

Select the Cache Provider and configure it in a hibernate.cfg.xml file. To do so

Use hibernate.cache.provider\_class property for Hibernate 3.

Use hibernate.cache.region.factory\_class  property cxif you are using hibernate 4

Each cache provider has a configuration file where we need to configure the cache configuration.

Enable Caching at a class or collection level in hbm xml file.

Add required jar files in the build path.

We will use ehcache provider to understand the caching.

**19.9.1 Configure EHCache Provider**

To configure EHCahce , add below lines in hibernate.cfg.xml.

**for Hibernate3**

   <property name="hibernate.cache.provider\_class">

        org.hibernate.cache.EhCacheProvider

   </property>

**for Hibernate 4**

  <property name="hibernate.cache.region.factory\_class">

        org.hibernate.cache.ehcache.EhCacheRegionFactory

  </property>

To use the singleton factory class,  use org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory class.

**19.9.2 EHCache Confguration**

We can configure the location of the ehcache configuration file in hibernate.cfg.xml file using net.sf.ehcache.configurationResourceName property. If we do not define then by default it will look for ehcache.xml file in the root directory. Below configuration in hibernate.cfg file will load myehcache.xml file available in the src root directory.

Note: If we don’t create ehcache.xml , it will not throw an error because it is available with default configuration in ehcache.jar file so that file will be picked.

<property name="net.sf.ehcache.configurationResourceName">

     /myehcache.xml

</property>

EHCache Configuration looks like below

|  |  |
| --- | --- |
| [?](http://www.wideskills.com/hibernate/hibernate-caching)5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | <?xml version="1.0" encoding="UTF-8"?>  <ehcache xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"      xsi:noNamespaceSchemaLocation="ehcache.xsd">     <diskStore path="java.io.tmpdir/ehcache" />     <defaultCache       maxElementsInMemory="500"       eternal="false"       timeToIdleSeconds="220"       timeToLiveSeconds="320"       overflowToDisk="true"   />      <cache name="com.tutorial.hibernate.Student"        maxElementsInMemory="500"        eternal="false"        timeToIdleSeconds="1800"        timeToLiveSeconds="2400"      overflowToDisk="false"    />  </ehcache> |

Details

**diskStore**:  EHCache does cache data in memory  and this property is used to specify the location where EHCache will store the data when it starts overflowing from memory.

**defaultCache**: This configuration will be used there is no configuration defined for an object which needs to be cached. This is a  mandatory configuration.

**cache name**=”com.tutorial.hibernate.Student” defines the cache region for Student class.

**eternal**=false means cache will be evicted/expire based on time

**timeToIdleSeconds**- specifies the expiry time in seconds the object is not accessed.

**timeToLiveSeconds**- specify the maximum time in seconds the object is will be available in cache

**maxElementsInMemory**- is the possible maximum elements of this class in cache.

**overflowToDisk**= enable or disable writing to disk when memory overflows.

So the above Student Cache configurations says to expire the cache if  an element is not accessed for 3 mins OR number of elements exceeds 500 or element is there in a cache for 4 mins.

**19.9.3 Enable Caching at a class level**

To do so add  <cache usage="read-only" />   in class mapping (hbm) . Usage attribute defines the concurrency strategy

**19.9.4  Add Required jar files**

Copy the below three jar files from the optional/ehcache directory of hibernate download and add it in a build path.(refer Chapter 3 for more details).

hibernate-ehcache-4.3.7.Final.jar

slf4j-api-1.6.1.jar

ehcache-core-2.4.3.jar

**19.10 Example – Second Level Cache**

Let’s play with ehcache in our example.

**Student Entity**

[?](http://www.wideskills.com/hibernate/hibernate-caching)

|  |  |
| --- | --- |
|  | package com.tutorial.hibernate;  public class Student {       private int id;       private String name;} |

**Hibernate.cfg.xml**

|  |  |
| --- | --- |
|  | <?xml version='1.0' encoding='utf-8'?>  <!DOCTYPE hibernate-configuration PUBLIC      "-//Hibernate/Hibernate Configuration DTD//EN"      "<http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd>">   <hibernate-configuration>     <session-factory>       <property name="hibernate.connection.url">            jdbc:mysql://localhost:3306/tutorial        </property>        <property name="hibernate.connection.username">          root        </property>        <property name="hibernate.connection.password">          password        </property>            <property name="dialect">org.hibernate.dialect.MySQLDialect</property>             <property name="hibernate.format\_sql">true</property>            <property name="show\_sql">true</property>            <property name="hibernate.cache.region.factory\_class">              org.hibernate.cache.ehcache.EhCacheRegionFactory            </property>             <property name="net.sf.ehcache.configurationResourceName">                  /myehcache.xml            </property>             <property name="hibernate.connection.driver\_class">              com.mysql.jdbc.Driver            </property>        <mapping resource="student.hbm.xml" />     </session-factory>  </hibernate-configuration> |

**student.hbm.xml**

|  |  |
| --- | --- |
|  | <hibernate-mapping package="com.tutorial.hibernate">     <class name="Student" table="student"  >        <cache usage="read-only" />        <id name="id" type="int" column="id">           <generator class="native"/>        </id>        <property name="name" column="name" type="string"/>     </class>  </hibernate-mapping> |

**Myehcahce.xml**

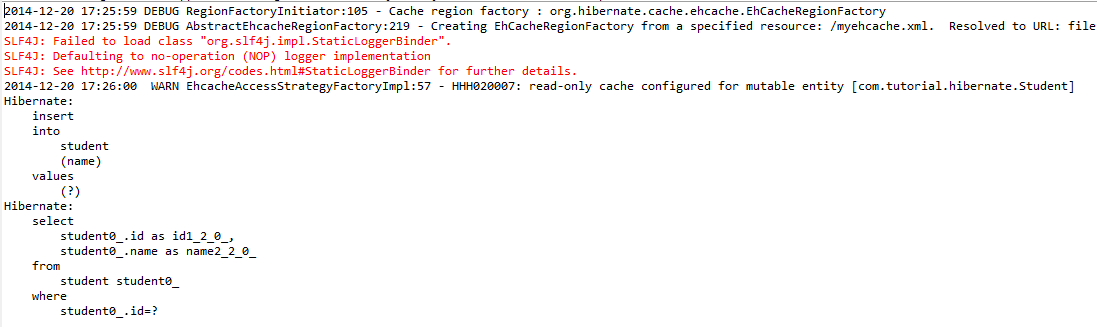
|  |  |
| --- | --- |
| [?](http://www.wideskills.com/hibernate/hibernate-caching) | <diskStore path="java.io.tmpdir/ehcache" />        <defaultCache maxElementsInMemory="500" eternal="false"      timeToIdleSeconds="220" timeToLiveSeconds="320" overflowToDisk="true" />       <cache name="com.tutorial.hibernate.Student" maxElementsInMemory="500"      eternal="false" timeToIdleSeconds="1800" timeToLiveSeconds="2400"      overflowToDisk="false" />  </ehcache> |

**Test Program**

|  |  |
| --- | --- |
|  | public class Test {         public static void main(String args[])       {            Configuration cfg = new Configuration().configure();            SessionFactory factory = cfg.buildSessionFactory();            Session session=  factory.openSession();              Transaction tx = session.beginTransaction();            Student student = new Student();            student.setName("Student A");              int id = (Integer)session.save(student);              tx.commit();            session.close();              session=  factory.openSession();              Student st1 = (Student) session.get(Student.class, id);            Student st2 = (Student) session.get(Student.class, id);            session.close();              session=  factory.openSession();              Student st3 = (Student) session.get(Student.class, id);            session.close();            factory.close();      }  } |

**Run Test Program**

Only one select statement is fired even though we fetched it three times (in different session also)

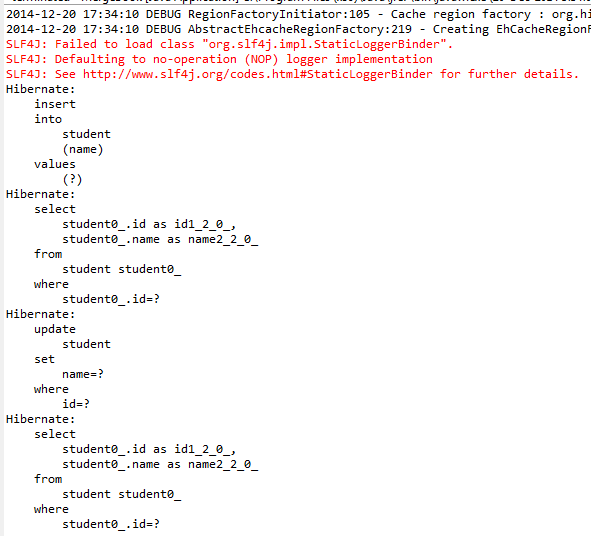


We configured **read-only** concurrency strategy so  if you will try to update the student like below, you will get an error. To solve this you can configure read-write  strategy

    st1.setName("XYZ");

    session.flush();

By doing so , you will see an extra update statement (we are updating student) and extra select because data has been changed in database



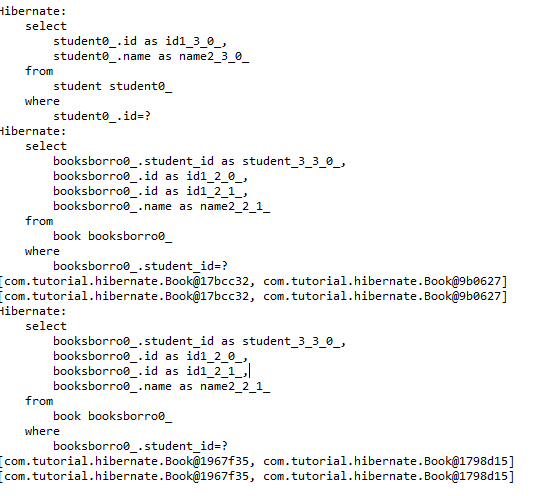
**19.11 – Caching Associations**

If a parent entity is cached, it will not cache the child objects. We need to turn on the caching of child entities.

Fetching associations in different session instead of having a parent object cached will fire select query to fetch the association.

If Student entity has a collections of books borrowed then fetching books like below will fire select statement on books for second session . This will be fired once per session because second time it will be available through the first level cache.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | public class Test {       public static void main(String args[])     {          Configuration cfg = new Configuration().configure();          SessionFactory factory = cfg.buildSessionFactory();          Session session=  factory.openSession();          Transaction tx = session.beginTransaction();            Book book1= new Book();          book1.setName("Book1");          Book book2= new Book();          book2.setName("Book2");          Set<Book> books = new HashSet<Book>();          books.add(book1);          books.add(book2);          session.save(book1);          session.save(book2);          Student student = new Student();          student.setName("Student A");          student.setBooksBorrowed(books);          int id = (Integer)session.save(student);          tx.commit();          session.close();          session=  factory.openSession();          Student st1 = (Student) session.get(Student.class, id);          Student st2 = (Student) session.get(Student.class, id);           System.out.println(st1.getBooksBorrowed());          System.out.println(st2.getBooksBorrowed());            session.close();           session=  factory.openSession();          Student st3 = (Student) session.get(Student.class, id);          System.out.println(st3.getBooksBorrowed());          Student st4 = (Student) session.get(Student.class, id);          System.out.println(st4.getBooksBorrowed());          session.close();          factory.close();    }} |
|  | <hibernate-mapping package="com.tutorial.hibernate">     <class name="Student" table="student"  >          <cache usage="read-write" />            <id name="id" type="int" column="id">              <generator class="native"/>            </id>          <property name="name" column="name" type="string"/>          <set name="booksBorrowed" >             <cache usage="read-write" />                 <key column="student\_id"/>                 <one-to-many class="Book"/>           </set>      </class>      <class name="Book" table="book"  >         <cache usage="read-write" />           <id name="isbn" type="int" column="id">              <generator class="native"/>           </id>           <property name="name" column="name" type="string"/>       </class>  </hibernate-mapping> |



If the collection is an entity, then object is cached with identifier (will have specific cache region with class name together with property) and if it is a value type then values are cached.

We need to enable caching at a Book level and also at a collection level

If we set caching in Book class as well then only one select for the book will be fired.

Adding <cache usage=”read-only”/ > at a Book level and in <set> will fire one select query on Book.

Need to add cache configuration in myehcache.xml as well.

**19.12 Query Level Cache**

By default the data of  HQL queries are not cached. If application fires HQL statement in the same or different session, multiple times, then data of the query can be cached.

To do so we need to enable, we need to first set **hibernate.cache.use\_query\_cache="true**" in hibernate.cfg.xml file . Setting up this property will create additional two required cache regions. One to store the data and one to hold the last updated timestamps.

Once done, we need to explicitly call setCacheable(true) on Query object before execution

By default Query results are cached with the region name “**org.hibernate.cache.QueryCache”**and we can change it by calling setCacheRegion(“Region Name”) on Query Object.

**Caching Annotations**

@Cacheable : triggers cache population

@CacheEvict : triggers cache eviction

@CachePut : updates the cache without interfering with the method execution

@Caching : regroups multiple cache operations to be applied on a method

@CacheConfig : shares some common cache-related settings at class-level

@EnableCaching : Configuration level annotation which enables Caching

In this post, we will learn the basics of caching and commonly used annotations including @EnableCaching, @Cacheable & @CacheEvict using EhCache as caching provider.

With Caching, using EHCache

Spring provides Caching abstractions and annotations to seamlessly adding caching support in any spring application. Main caching annotations being used are @EnableCaching, @Cacheable & @CacheEvict.

1. @Cacheable [ Used for populating and accessing Cache]

@Cacheable annotation indicates that the result of invoking a method (or all methods in a class) can be cached. A cache itself can be imagined as a key-value based Map. First time a method annotated with @Cacheable gets called, it gets executed and it’s return value is stored in Cache using a key[method parameter for instance, ]. Next time, if the method gets called using same key[same parameter for instance], the result is returned directly from Cache, method invocation does not take place.

By default it uses the method parameters [product name in this case] to compute the key , but a SpEL expression can be provided via the key() attribute, or a custom [KeyGenerator](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/cache/interceptor/KeyGenerator.html) implementation can replace the default one. Check out the [Official reference](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/cache/annotation/Cacheable.html) for details of all possible attributes.

|  |
| --- |
| package com.websystique.spring.service;  import org.springframework.cache.annotation.Cacheable;  import org.springframework.stereotype.Service;   @Service("productService")  public class ProductServiceImpl implements ProductService{       @Override      @Cacheable("products")      public Product getByName(String name) {          slowLookupOperation();          return new Product(name,100);    }       @CacheEvict(value = "products", allEntries = true)      public void refreshAllProducts() {         //This method will remove all 'products' from cache, say as a result of flush API call.      }      public void slowLookupOperation(){           try {                  long time = 5000L;                  Thread.sleep(time);              } catch (InterruptedException e) {                  throw new IllegalStateException(e);             }    }} |

2. @CacheEvict [ Used for removing items from Cache]

@CacheEvict annotation indicates that a method (or all methods on a class) triggers a cache evict operation, removing specific[or all] items from cache. Various [attributes](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/cache/annotation/CacheEvict.html) provides complete control to enforce the required behavior for cache-eviction.

3. @EnableCaching [ Used for Enabling Caching support in Spring Applicaion]

@EnableCaching annotation triggers a post processor that inspects every Spring bean for the presence of caching annotations[@Cacheable, @CacheEvict, @CachePut] on public methods. If such an annotation is found, a proxy is automatically created to intercept the method call and handle the caching behavior accordingly.

**Evict Second Level Cache**To remove completely second level cache, we can use following codes:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | @Autowired  private SessionFactory sessionFactory;  public void evictAll() {  SessionFactory sf = currentSession().getSessionFactory();  Cache cache = sf.getCache();  cache.evictQueryRegions();  cache.evictDefaultQueryRegion();  cache.evictCollectionRegions();  cache.evictEntityRegions();  }  protected Session currentSession() {  return sessionFactory.getCurrentSession();  } |

If we want to **evict** single entity, all entities or collections from cache, we can use other **evict** methods as follows: 

|  |  |
| --- | --- |
| 1  2  3  4  5 | SessionFactory sf = currentSession().getSessionFactory();  cache.evictEntity(Cat.class, catId); //evict a particular Cat  cache.evictEntityRegion(Cat.class); //evict all Cats  cache.evictCollection("Cat.kittens", catId); //evict a particular collection of kittens  cache.evictCollectionRegion("Cat.kittens"); //evict all kitten collections |

First Level Cache  
Hibernate first level cache is session specific, so when calling same entity data in same session, there will be no hit to database. If you update the entity in the same session and calling it, we will see old data because this is default behavior of the first level cache. However in other session query we will see updated data.   
  
We can use session evict() method to remove a single object from the hibernate first level cache.   
We can use session clear() method to clear the cache i.e delete all the objects from the cache.   
We can use session contains() method to check if an object is present in the hibernate cache or not, if the object is found in cache, it returns true or else it returns false.   
Since hibernate cache all the objects into session first level cache, while running bulk queries or batch updates it’s necessary to clear the cache at certain intervals to avoid memory issues.   
**@Transactional Annotation**  
If this is the transaction boundary for your service layer then Hibernate will create a new Session, meaning there's nothing in the first level cache. If you try to call the findOne()method twice within the same service method, the second call will fetch the entity from the cache.   
Successive service method calls (e.g. getEntity) always end up with a new Hibernate Session, so a fresh entity is loaded from the database.   
If you employ a 2nd level cache and activate it for this entity, then Hibernate will always hit the cache first and fallback to database loading, on a cache miss. 

# Transactions and Concurrency in Hibernate

**20.1 Overview**

A transaction is a unit of work in which either all operations must execute or none of them. To understand the importance of transaction, think of a example which applies on all of us. “Transferring Amount from one account to another “ – this operations includes below at least below two steps

Deduct the balance from sender’s account

Add the amount to the receiver’s account.

Now think of the situation where amount is deducted from sender’s account but not gets delivered to receiver account due to some errors. Such issues are managed by transaction management in which both the steps are performed in a single unit of work where either both steps are performed successfully or in case anyone gets failed, it should be roll backed.

There are four important terms which are very important to understand.

Atomic -  As described above , atomicity makes sure that either all operations within a transactions must be successful or none of them.

Consistent- This property makes sure that data should be in consistent state once the transaction is completed.

Isolated-  this property allows multiple users to access the same set of data and each user’s processing should be isolated from others.

Durable – Result of the transaction should be permanent once transaction is completed to avoid any loss of data.

**20.2 Hibernate Transaction Management**

It is very important to understand the difference between transaction boundaries and transaction demarcation. Starting and end point of a transaction are known as transaction boundaries and technique to identify transaction boundaries are known as transaction demarcation.

Database operations can be performed simultaneously (concurrently) by multiple users and this can end up with serious impacts. In order to control the concurrency , there are two approaches

Optimistic  - Versioning is used in this approach.

Pessimistic – Acquiring Lock mechanism is used in this approach.

**20.3 Programmatic Transaction Demarcation**

We can use Core JDBC  transactions by Connection (java,sql.Connection) object . We need to disable the auto commit of connection by using setAutoCommit(false) method on JDBC connection. Doing so, JDBC will not commit the data updates and we have to explicitly call commit() and rollback() methods to commit and rollback the changes.

Hibernated comes with own transaction management (org.hibernate.Transaction) which we have used in all examples we have discussed so far in this tutorial.

If the transaction includes multiple data sources or resources then core JDBC transaction will not suffice and we would need to include Transaction manager which has a capability to commit  the  changes in all data sources in all updates are successful and rolls back if anyone gets failed. This type of transaction management, uses Java Transaction API (JTA) which exposes UserTransaction (javax.transaction.UserTransaction) interface.

**Note:  D**atabase connection handling is different between JDBC and JTA transactions. Hibernate gets a connection for every Session and  tries to be as lazy as possible.

Without JTA, Hibernate would hold on to a particular database connection from the beginning until the end of the transaction.

With a JTA configuration, a connection is obtained and used for only a single SQL statement and then is immediately returned back to the connection pool. The application server guarantees that it will provide the same connection during the same transaction, when ever needed for another SQL statement.

**20.4 Connection Pooling**

We would need connection pooling if application is being used by a multiple users. Multiple connections would be needed. Creating a connection every time is a very expensive operation so we should use connection pooling.

In a web application , application server can manage the connection pool but for standalone we have to rely on some third party solutions.

We can integrate Apache’s connection pooling solutions (DBCP) or can use the C3P0 pooling framework with Hibernate. In fact Hibernate comes with C3P0.

**20.4.1 Enabling Connection Pooling in Stand Alone Hibernate Application**

Download the connection pool framework jar file and add it in a build path. You can get the required jar files for C3P0 from lib/optional directory of hibernate download.

Once done we need to configure the connection pools properties in hibernate.cfg.xml like below

       <property name="connection.provider\_class">org.hibernate.connection.C3P0ConnectionProvider" />

       <property name="hibernate.c3p0.min\_size">15</property>

       <property name="hibernate.c3p0.max\_size">30</property>

      <property name="hibernate.c3p0.timeout">300</property>

       <property name="hibernate.c3p0.acquire\_increment">1</property>

hibernate.connection.provider\_class is used to specify the Connection Provider class.

hibernate.c3p0.min\_size specifies the minimum connections that will be ready to serve the request every time.

hibernate.c3p0.max\_size specifies the maximum number of connections in a pool. This is mandatory property.

hibernate.c3p0.timeout specifies the max idle time for a connection. After this connection will be removed from the pool.

hibernate.c3p0.acquire\_increment - is the number of connections created in the pool when all connections in the pool get exhausted.

**20.5 Programmatic Transaction in Stand Alone (Non Managed) Hibernate Application**

Default transaction factory class for Hibernate is **JDBCTransactionFactory** and we can configure it using **hibernate.transaction.factory\_class**property. Responsibility of Transaction factory is to instantiate transactions.

Transaction in hibernate standalone application we need to follow below steps

get Session Object from SessionFactory.openSession() . Remember opening a session will not open a connection with database which means opening a session is not expensive.

Get org.hibernate.Transaction object  from session.beginTransaction()  . This step opens the connection with database. Internally this call disables the auto commit of underlying connection.

Call transaction,commit() to persist the changes in database.

Call  transaction.rollback() to rollback the changes. Usually done in case of exceptions (catch block)

[?](http://www.wideskills.com/hibernate/transactions-and-concurrency-in-hibernate)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | Configuration cfg = new Configuration().configure();  factory = cfg.buildSessionFactory();      Session session =factory.openSession();      Transaction tx = session.beginTransaction();      try{               session.save(entity);           tx.commit();         } catch(Exception e)             {                tx.rollback();             }                session.close(); |

**20.6 Programmatic Transaction in Managed Environment  using JTA**

Managed Environments are managed by the application servers like JBoss, Web sphere etc so this section is **more** applicable on the web applications. In upcoming chapters we will discuss using Hibernate in Web applications.

You can use open source JTA provider and integrate it with stand alone applications also.

Lets discuss how we can manage transaction and concurrency in managed environments.

a. We need to configure the transaction factory because default JDBC Transaction Factory will not work so we will need to configure it as **JTATransactionFactory**.

hibernate.transaction.factory\_class=org.hibernate.transaction.JTATransactionFactory

b. Now we need to inform Hibernate about the JTA implementation we are going to use for our application .As mentioned above it could be any open source JTA or we can use the ones which is provided by application servers in case of web applications. Hibernate supports most of the application server look ups.  To configure we need to use hibernate.transaction.manager\_lookup\_class property.

 hibernate.transaction.manager\_lookup\_class=org.hibernate.transaction.WeblogicTransactionManagerLookup

  For Joss we can use

 hibernate.transaction.manager\_lookup\_class= org.hibernate.transaction.JBossTransactionManagerLookup

As we mentioned in the initial chapters of this tutorial that the creation of SessionFactory object is very expensive and as it is thread safe we need one Session factory per database in entire application. In case of web application, we can create a utility class which will be responsible for creating the session factory and exposing it as static object.

Now,  as the transaction is JTA and it is not longer tied to any session so we cannot get it from  session object. Instead, we can grab the User Transaction object from JNDI as

UserTransaction tx = (UserTransaction)new InitialContext()

.lookup("java:comp/UserTransaction");

Now tx can be used in the same way it is being used in standalone application and we this transaction we can manage multiple sessions  like below pseudo code

|  |  |
| --- | --- |
| 17  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | Configuration cfg1 = new Configuration().configure();  factory1 = cfg1.buildSessionFactory();  Session session1 =factory1.openSession();  Configuration cfg2 = new Configuration().configure();  factory2 = cfg2.buildSessionFactory();  Session session2 =factory2.openSession();  UserTransaction tx = (UserTransaction)new InitialContext()  .lookup("java:comp/UserTransaction");  try{     session1.save(entity);     session2.save(entity);      tx.commit();      } catch(Exception e)    {          tx.rollback();}  Session1.close();  Session2.close(); |

**20.7 Controlling Concurrent Access**

Isolation one of the ACID property which defines when the changes done by a transaction will be visible to other transactions. Each database vendor provides a transaction isolation level (each transaction assumes that no other transaction is going on) using locking approach (put a lock on the data so no other transaction can access that piece of data and release the lock once done). ORM frameworks like Hibernate improves the transaction isolation level provided by the database with which application is interacting with.

It is important to know that hibernate does not lock anything in memory.

Achieving complete Isolation comes with a price of scalability and performance where as weakening the  isolation level helps in improving performance. With this statement, it is clear that there is no 0 or 1 situation with isolation (either isolation is there or none). Instead there are several level of isolation are supported.

**20.7.1 Problem or Issues with  concurrent data access.**

Below are the scenarios that can occur with the impact of concurrent data access.

**Lost Updates** – one transaction (T1) updates the data and commits successfully where as the second transaction (T2) fails to commit. In this case updates done by transaction (T1) are lost. This can be the situation when concurrent transactions are not isolated.

**Dirty Read (Read Uncommitted Data)** -  One transaction (T1) reads the uncommitted data (updates are done, but not committed) by another transaction (T2).

**Phantom Read**-  Transaction (T1) executes same query twice and the result sets is different each time, probably because another transaction (T2) has either added or deleted the records.

**Unrepeatable Read**- Transaction (T1) reads the same row twice and state of row is different probably because another transaction (T2) updates the row. Special case would be if T1 updates and committed the same data again, then updates done by T2 will be lost.

**20.7.2 Transaction Isolation Levels**

 As we mentioned that isolation does not have  0 or 1 state instead there can be different isolation levels. I would like to reiterate that increasing the isolation level will impact the performance. Lets discuss the isolation levels (degree of level on locking the data) and which isolation level can prevent  which issue discussed in section 20.7.1

JTA defines the same isolation level as ANSI –SQL standard does. There are four transaction isolation levels.

**Read Uncommitted-**One transaction can view uncommitted data of another transaction and dirty read, phantom read, unrepeatable reads are allowed. This is the loosest isolation level and is not recommended.

**Read Committed** – Dirty Reads (Uncommitted Read) are not allowed in this isolation level, but unrepeatable reads and phantom reads are permitted. This approach uses shared read lock and exclusive write lock in which read lock is acquired and released immediately where as write lock is released at the end of the transaction.

**Repeatable Read-**  Dirty Read and Unrepeatable Read is not allowed in this isolation level but phantom reads are allowed. In this isolation level, reading transaction will block all other writing transactions ( but allows other reading transactions ) and any writing transaction will block all other transactions. This will have some scalability issues.

**Serializable-**This is the strictest isolation level and will have  scalability issues. This prevents dirty read, phantom reads , unrepeatable read etc. Transactions are executed serially (one after another) and acquires read and write locks on the entire set of affected data.

|  |  |  |  |
| --- | --- | --- | --- |
| **Isolation Level** | **Phantom Read** | **Unrepeatable Read** | **Dirty Read** |
| Read Uncommitted | Allowed | Allowed | Allowed |
| Read Committed | Allowed | Allowed | Not Allowed |
| Repeatable Read | Allowed | Not Allowed | Not Allowed |
| Serializable | Not allowed | Not allowed | Not allowed |

**20.7.3 How to set a Transaction Isolation level**

Every database has default isolation level (either Read Committed or Repeatable Read) and we can override the isolation level using **hibernate.connection.isolation**property in hibernate.cfg.xml file.

Read uncommitted isolation - 1

Read committed isolation- 2

Repeatable read isolation- 4

Serializable isolation – 8

**20.7.4 Enable Versioning in Hibernate**

Hibernate has a inbuilt auto versioning of type integer or timestamp at a entity level. To enable versioning , we need to add a property of type int or integer in entity and configuration in mapping of corresponding entity using <version> tag.

Below sample configuration says that there is a property with name version-field in java entity and it is mapped to version column of table.

         <version name="version-field" access="field" column="version"></version>

To use version field as timestamp, define property of data type date and corresponding mapping using <timestamp> tag

        <timestamp name="version-field" access="field" column="timestamp"></timestamp>

Difference between local and global transaction ?

* **Local Transactions** - A local transaction is internal to a single resource.
* **Global Transactions** - A global transaction is created by an external transaction manager (JTA) and is used to scope work on multiple resources.

ocal transactions are transactions associated with aparticalar data source (means they are resource-specific).the most common example would be a transaction associatedwith a JDBC connection. While Global Transactions providethe ability to work with multiple transactional resources  
(typically relational databases and message queues).

Example of Local transaction :  
<bean id="dataSource"  
class="org.apache.commons.dbcp.BasicDataSource"  
destroy-method="close">  
<property name="driverClassName"  
value="${jdbc.driverClassName}" />  
<property name="url" value="${jdbc.url}" />  
<property name="username" value="${jdbc.username}" />  
<property name="password" value="${jdbc.password}" />  
</bean>  
<bean id="txManager"  
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
<property name="dataSource" ref="dataSource"/>  
</bean>  
DataSourceTransactionManager : takes the datasource as oneof its properties.  
even HibarnateTransactionManager takes sessionfactory whichin turn uses datasource as a property or it also takes datasource as one of the property.  
Example of Global transaction :   
If we use JTA in a J2EE container, as in the'dataAccessContext-jta.xml' file from the same sample  
application, we use a container DataSource, obtained viaJNDI, in conjunction with Spring'sJtaTransactionManager. The JtaTransactionManager doesn'tneed to know about the DataSource, or anyother specific resources, as it will use the container'sglobal transaction management infrastructure.  
<?xml version="1.0" encoding="UTF-8"?>  
<beans xmlns="http://www.springframework.org/schema/beans"  
 ----\*\*\*\*2.5.xsd">  
<jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>  
<bean id="txManager"  
class="org.springframework.transaction.jta.JtaTransactionManager"  
/>  
<!-- other <bean/> definitions here -->  
</beans>