FAQ : Hibernate Interview Questions

1.What is Hibernate?
Hibernate is a pure Java object-relational mapping (ORM) and persistence framework that allows you to map plain old Java objects to relational database tables using (XML) configuration files. Its purpose is to relieve the developer from a significant amount of relational data persistence-related programming tasks. This lets the users to develop persistent classes following object-oriented principles such as association, inheritance, polymorphism, composition, and collections.

2.What is ORM?
ORM stands for Object/Relational mapping. It is the programmed and translucent perseverance of objects in a Java application in to the tables of a relational database using the metadata that describes the mapping between the objects and the database. It works by transforming the data from one representation to another.

3.What does an ORM solution comprises of?

- It should have an API for performing basic CRUD (Create, Read, Update, Delete) operations on objects of persistent classes
- Should have a language or an API for specifying queries that refer to the classes and the properties of classes
- An ability for specifying mapping metadata
- It should have a technique for ORM implementation to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions

4.What are the different levels of ORM quality?
There are four levels defined for ORM quality.
i. Pure relational
ii. Light object mapping
iii. Medium object mapping
iv. Full object mapping

5.What is a pure relational ORM?
The entire application, including the user interface, is designed around the relational model and SQL-based relational operations.

6.What is a meant by light object mapping?
The entities are represented as classes that are mapped manually to the relational tables. The code is hidden from the business logic using specific design patterns. This approach is successful for applications with a less number of entities, or applications with common, metadata-driven data models. This approach is most known to all.

7.What is a meant by medium object mapping?
The application is designed around an object model. The SQL code is generated at build time. And the associations between objects are supported by the persistence mechanism, and queries are specified using an object-oriented expression language. This is best suited for medium-sized applications with some complex transactions. Used when the mapping exceeds 25 different database products at a time.

8.What is meant by full object mapping?
Full object mapping supports sophisticated object modeling: **composition, inheritance, polymorphism and persistence**. The persistence layer implements transparent persistence; persistent classes do not inherit any special base class or have to implement a special interface. Efficient **fetching strategies and caching strategies** are implemented transparently to the application.

9.What are the benefits of ORM and Hibernate?
There are many benefits from these. Out of which the following are the most important one.

i. Productivity - Hibernate reduces the burden of developer by providing much of the functionality and let the developer to concentrate on business logic.

ii. Maintainability - As hibernate provides most of the functionality, the LOC for the application will be reduced and it is easy to maintain. By automated object/relational persistence it even reduces the LOC.

iii. Performance - Hand-coded persistence provided greater performance than automated one. But this is not true all the times. But in hibernate, it provides more optimization that works all the time there by increasing the performance. If it is automated persistence then it still increases the performance.

iv. Vendor independence - Irrespective of the different types of databases that are there, hibernate provides a much easier way to develop a cross platform application.

10.Why do you need ORM tools like hibernate?
The main advantage of ORM like hibernate is that it shields developers from messy SQL. Apart from this, ORM provides following benefits:

* Improved productivity
High-level object-oriented API
Less Java code to write
No SQL to write

* Improved performance
Sophisticated caching
Eager loading

* Improved maintainability
A lot less code to write

* Improved portability
ORM framework generates database-specific SQL for you

11.What is object/relational mapping metadata?
ORM tools require a metadata format for the application to specify the mapping between classes and tables, properties and columns, associations and foreign keys, Java types and SQL types. This information is called the object/relational mapping metadata. It defines the transformation between the different data type systems and relationship representations.

12.Why Hibernate is better than JDBC? (or) What is the difference between hibernate and jdbc ? (or) what are the advantages of Hibernate over jdbc?

Advantage of Hibernate over JDBC
-Hibernate provides database independent query mechanism but JDBC does not. You need to write database specific query.

-Hibernate treats tables as objects and so you can work with classes and objects instead of queries and result sets but its not so with jdbc.

-Hibernate provides its own powerful query language called HQL (Hibernate Query Language). Using HQL you can express queries in a familiar SQL like syntax. It selects the best way to execute the database operations. JDBC supports only native Structured Query Language (SQL) and you need to write the effective query to access database.

- 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.

-Hibernate eliminates need for repetitive SQL.
-You will get benefit of Cache. Hibernate support 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.

-Hibernate supports query caching for better performance. Hibernate supports Query cache and It will provide the statistics about your query and database status.
JDBC Not provides any statistics.

- No need to create any connection pool in case of Hibernate. Hibernate provides connection pooling by just mentioning a few lines in configuration file. You can use c3p0. In case of JDBC you need to write your own connection pool

-XML file in hibernate lets you specify all configuration information and relations between tables , which increases the readability and allows changes more easily.

-Hibernate supports lazy loading. Objects can also be loaded on startup by turning off the lazy attribute. Jbdc does not support it.

-Hibernate supports Automatic Versioning and Time Stamping but jdbc does not.

-Hibernate provides API to handle all create-read-update-delete (CRUD) operations i.e. no SQL is required.

-Hibernate makes the development fast because it relieves you from manual handling of persistent data.

13.What the are Core interfaces of hibernate framework?
There are many benefits from these. Out of which the following are the most important one.

i. Session Interface - This is the primary interface used by hibernate applications. The instances of this interface are lightweight and are inexpensive to create and destroy. Hibernate sessions are not thread safe.

ii. SessionFactory Interface - This is a Factory that delivers the session objects to hibernate application. Generally there will be a single SessionFactory for the whole application and it will be shared among all the application threads.

iii. Configuration Interface - This interface is used to configure and bootstrap hibernate. The instance of this interface is used by the application in order to specify the location of hibernate specific mapping documents.

iv. Transaction Interface - This is an optional interface but the above three interfaces are mandatory in each and every application. This interface abstracts the code from any kind of transaction implementations such as JDBC transaction, JTA transaction.

v. Query and Criteria Interface - This interface allows the user to perform queries and also control the flow of the query execution.

14.What are Callback interfaces?
These interfaces are used in the application to receive a notification when some object events occur. Like when an object is loaded, saved or deleted. There is no need to implement callbacks in hibernate applications, but they're useful for implementing certain kinds of generic functionality.

15.What are Extension interfaces?
When the built-in functionalities provided by hibernate is not sufficient enough, it provides a way so that user can include other interfaces and implement those interfaces for user desire functionality. These interfaces are called as Extension interfaces.

16.What are the Extension interfaces that are there in hibernate?
There are many extension interfaces provided by hibernate.

ProxyFactory interface - used to create proxies

ConnectionProvider interface - used for JDBC connection management

TransactionFactory interface - Used for transaction management

Transaction interface - Used for transaction management

TransactionManagementLookup interface - Used in transaction management.

Cache interface - provides caching techniques and strategies

CacheProvider interface - same as Cache interface

ClassPersister interface - provides ORM strategies

IdentifierGenerator interface - used for primary key generation

Dialect abstract class - provides SQL support

17.What are different environments to configure hibernate?
There are mainly two types of environments in which the configuration of hibernate application differs.

i. Managed environment - In this kind of environment everything from database connections, transaction boundaries, security levels and all are defined. An example of this kind of environment is environment provided by application servers such as JBoss, Weblogic and WebSphere.

ii. Non-managed environment - This kind of environment provides a basic configuration template. Tomcat is one of the best examples that provide this kind of environment.

18.How does hibernate code looks like?
```
Session session = getSessionFactory().openSession();
Transaction tx = session.beginTransaction();
MyPersistanceClass mpc = new MyPersistanceClass ("Sample App");
session.save(mpc);
tx.commit();
session.close();
```

19.What is a hibernate xml mapping document and how does it look like?
In order to make most of the things work in hibernate, usually the information is provided in an xml document. This document is called as xml mapping document. The document defines, among other things, how properties of the user defined persistence classes' map to the columns of the relative tables in database.
```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
<class name="sample.MyPersistanceClass" table="MyPersitaceTable">
<id name="id" column="MyPerId">
<generator class="increment"/>
</id>
<property name="text" column="Persistance_message"/>
<many-to-one name="nxtPer" cascade="all" column="NxtPerId"/>
</class>
</hibernate-mapping>
```

Everything should be included under <hibernate-mapping> tag. This is the main tag for an xml mapping document.

20.What is the file extension you use for hibernate mapping file?
The name of the file should be like this : filename.hbm.xml
The filename varies here. The extension of these files should be ".hbm.xml".
This is just a convention and it's not mandatory. But this is the best practice to follow this extension.

21.How do you create a SessionFactory?

```
Configuration cfg = new Configuration();
cfg.addResource("myinstance/MyConfig.hbm.xml");
cfg.setProperties( System.getProperties() );
SessionFactory sessions = cfg.buildSessionFactory();
```

First, we need to create an instance of Configuration and use that instance to refer to the location of the configuration file. After configuring, this instance is used to create the SessionFactory by calling the method buildSessionFactory().

22.What is meant by Method chaining?
Method chaining is a programming technique that is supported by many hibernate interfaces. This is less readable when compared to actual java code. And it is not mandatory to use this format. Look how a SessionFactory is created when we use method chaining.

```
SessionFactory sf = new Configuration()
        .addResource("myinstance/MyConfig.hbm.xml")
        .setProperties( System.getProperties() )
        .buildSessionFactory();
```

23.What does hibernate.properties file consist of?
This is a property file that should be placed in application class path. So when the Configuration object is created, hibernate is first initialized. At this moment the application will automatically detect and read this hibernate.properties file.

```
hibernate.connection.datasource = java:/comp/env/jdbc/AuctionDB
hibernate.transaction.factory_class =net.sf.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class =
net.sf.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = net.sf.hibernate.dialect.PostgreSQLDialect
```

24.Where should SessionFactory be placed so that it can be easily accessed?
As per J2EE environment, if the SessionFactory is placed in JNDI then it can be easily accessed and shared between different threads and various components that are hibernate aware. You can set the SessionFactory to a JNDI by configuring a property **hibernate.session_factory_name** in the **hibernate.properties** file or **hibernate.cfg.xml** file.

25.What are POJOs? *IMP*
POJO stands for plain old java objects. These are just basic JavaBeans that have defined setter and getter methods for all the properties that are there in that bean. Besides they can also have some business logic related to that property. Hibernate applications works efficiently with POJOs rather then simple java classes.

26.What is HQL?
Hibernate Query Language. Hibernate allows the user to express queries in its own portable SQL extension and this is called as HQL. It also allows the user to express in native SQL.

27.What are the different types of property and class mappings?
Typical and most common property mapping

```
<property  name="description" column="DESCRIPTION" type="string"/>
```

Or

```
<property name="description" type="string">
        <column name="DESCRIPTION"/>
</property>
```

Derived properties

```
<property name="averageBidAmount"
        formula="( select AVG(b.AMOUNT) from BID b where b.ITEM_ID = ITEM_ID )"
type="big_decimal"/>
```

Controlling inserts and updates

```
<property name="name" column="NAME" type="string" insert="false" update="false"/>
```

28.What is Attribute Oriented Programming?
XDoclet has brought the concept of attribute-oriented programming to Java. Until JDK 1.5, the Java language had no support for annotations; now XDoclet uses the Javadoc tag format (@attribute) to specify class-, field-, or method-level metadata attributes. These attributes are used to generate hibernate mapping file automatically when the application is built. This kind of programming that works on attributes is called as Attribute Oriented Programming.

29.What are the different methods of identifying an object?
There are three methods by which an object can be identified.
i. Object identity -Objects are identical if they reside in the same memory location in the JVM. This can be checked by using the = = operator.

ii. Object equality - Objects are equal if they have the same value, as defined by the equals () method. Classes that don't explicitly override this method inherit the implementation defined by java.lang.Object, which compares object identity.

iii. Database identity - Objects stored in a relational database are identical if they represent the same row or, equivalently, share the same table and primary key value.

30.What are the benefits of inheritance?
1. Code reusebility.
2. Minimise the amount of duplicate code.
3. Sharing common code amongst several subclasses.
4. Smaller, simpler and better organisation of code.
5. Make application code more flexible to change.

31.What are the different approaches to represent an inheritance hierarchy in hibernate?
Table per concrete class.
Table per class hierarchy.
Table per subclass.

32.What are managed associations and hibernate associations?
Associations that are related to container management persistence are called managed associations. These are bi-directional associations. Coming to hibernate associations, these are unidirectional.

Other FAQS

***Q1*.How can I count the number of query results without actually returning them?**

```
Integer count = (Integer) session.createQuery("select count(*) from ....").uniqueResult();
```

Q2.How can I find the size of a collection without initializing it?

```
Integer size = (Integer) s.createFilter( collection, "select count(*)" ).uniqueResult();
```

Q3.How can I order by the size of a collection?

Use a left join, together with group by

```
select user
from User user
left join user.messages msg
group by user
order by count(msg)
```

Q4.How can I place a condition upon a collection size?

If your database supports subselects:

```
from User user where size(user.messages) >= 1
```

or:

```
from User user where exists elements(user.messages)
```

If not, and in the case of a one-to-many or many-to-many association:

```
select user
from User user
join user.messages msg
group by user
having count(msg) >= 1
```

Because of the inner join, this form can't be used to return a User with zero messages, so the following form is also useful

```
select user
from User as user
left join user.messages as msg
group by user
having count(msg) = 0
```

Q4.How can I query for entities with empty collections?

```
from Box box
where box.balls is empty
```

Or, try this:

```
select box
from Box box
  left join box.balls ball
where ball is null
```

Q5.How can I sort (or) order collection elements?
There are three different approaches:
1. Use a SortedSet or SortedMap, specifying a comparator class in the sort attribute or <set> or <map>. This solution does a sort in memory.

2. Specify an order-by attribute of <set>, <map> or <bag>, naming a list of table columns to sort by. This solution works only in JDK 1.4+.

3. Use a filter session.createFilter(collection, "order by ....").list()

Q6.Are collections pageable?

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

Q7.I have a one-to-one association between two classes. Ensuring that associated objects have matching identifiers is bugprone. Is there a better way?

```
<generator class="foreign">
    <param name="property">parent</param>
</generator>
```

Q8.I have a many-to-many association between two tables, but the association table has some extra columns (apart from the foreign keys). What kind of mapping should I use?

Use a composite-element to model the association table. For example, given the following association table:

```
create table relationship (
   fk_of_foo bigint not null,
   fk_of_bar bigint not null,
   multiplicity smallint,
   created date )
```

you could use this collection mapping (inside the mapping for class Foo):

```
<set name="relationship">
   <key column="fk_of_foo"/>
   <composite-element class="Relationship">
      <property name="multiplicity" type="short" not-null="true"/>
      <property name="created" type="date" not-null="true"/>
      <many-to-one name="bar" class="Bar" not-null="true"/>
   </composite-element>
</set>
```

You may also use an <idbag> with a surrogate key column for the collection table. This would allow you to have nullable columns.

An alternative approach is to simply map the association table as a normal entity class with two bidirectional one-to-many associations.

Q9.In an MVC application, how can we ensure that all proxies and lazy collections will be initialized when the view tries to access them?

One possible approach is to leave the session open (and transaction uncommitted) when forwarding to the view. The session/transaction would be closed/committed after the view is rendered in, for example, a servlet filter (another example would by to use the ModelLifetime.discard() callback in Maverick). One difficulty with this approach is making sure the session/transaction is closed/rolled back if an exception occurs rendering the view.

Another approach is to simply force initialization of all needed objects using Hibernate.initialize(). This is often more straightforward than it sounds.

Q10.How can I bind a dynamic list of values into an in query expression?

```
Query q = s.createQuery("from foo in class Foo where foo.id in (:id_list)");
q.setParameterList("id_list", fooIdList);
List foos = q.list();
```

Q11.How can I bind properties of a JavaBean to named query parameters?

```
Query q = s.createQuery("from foo in class Foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();
```

Q12.Can I map an inner class?

You may persist any static inner class. You should specify the class name using the standard form ie. eg.Foo$Bar.

Q13.How can I assign a default value to a property when the database column is null?

Use a UserType.

Q14.How can I truncate String data?

Use a UserType.

Q15.How can I trim spaces from String data persisted to a CHAR column?

Use a UserType.

Q16.How can I convert the type of a property to/from the database column type?

Use a UserType.

Q17.How can I get access to O/R mapping information such as table and column names at runtime?

This information is available via the Configuration object. For example, entity mappings may be obtained using Configuration.getClassMapping(). It is even possible to manipulate this metamodel at runtime and then build a new SessionFactory.

Q18.How can I create an association to an entity without fetching that entity from the database (if I know the identifier)?

If the entity is proxyable (lazy="true"), simply use load(). The following code does not result in any SELECT statement:

```
Item itemProxy = (Item) session.load(Item.class, itemId);
Bid bid = new Bid(user, amount, itemProxy);
session.save(bid);
```

Q18.How can I retrieve the identifier of an associated object, without fetching the association?

Just do it. The following code does not result in any SELECT statement, even if the item association is lazy.

```
Long itemId = bid.getItem().getId();
```

This works if getItem() returns a proxy and if you mapped the identifier property with regular accessor methods. If you enabled direct field access for the id of an Item, the Item proxy will be initialized if you call getId(). This method is then treated like any other business method of the proxy, initialization is required if it is called.

Q20.How can I manipulate mappings at runtime?

You can access (and modify) the Hibernate metamodel via the Configuration object, using getClassMapping(), getCollectionMapping(), etc.

Note that the SessionFactory is immutable and does not retain any reference to the Configuration instance, so you must re-build it if you wish to activate the modified mappings.

Q21.How can I avoid n+1 SQL SELECT queries when running a Hibernate query?

Follow the best practices guide! Ensure that all <class> and <collection> mappings specify lazy="true" in Hibernate2 (this is the new default in Hibernate3). Use HQL LEFT JOIN FETCH to specify which associations you need to be retrieved in the initial SQL SELECT.

A second way to avoid the n+1 selects problem is to use fetch="subselect" in Hibernate3.

If you are still unsure, refer to the Hibernate documentation and Hibernate in Action.

Q22.I have a collection with second-level cache enabled, and Hibernate retrieves the collection elements one at a time with a SQL query per element!

Enable second-level cache for the associated entity class. Don't cache collections of uncached entity types.

Q23.How can I insert XML data into Oracle using the xmltype() function?

Specify custom SQL INSERT (and UPDATE) statements using <sql-insert> and <sql-update> in Hibernate3, or using a custom persister in Hibernate 2.1.

You will also need to write a UserType to perform binding to/from the PreparedStatement.

Q24.How can I execute arbitrary SQL using Hibernate?

```
PreparedStatement ps = session.connection().prepareStatement(sqlString);
```

Or, if you wish to retrieve managed entity objects, use session.createSQLQuery().

Or, in Hibernate3, override generated SQL using <sql-insert>, <sql-update>, <sql-delete> and <loader> in the mapping document.

Q25.I want to call an SQL function from HQL, but the HQL parser does not recognize it!

Subclass your Dialect, and call registerFunction() from the constructor.

Q26. What are the most common methods of Hibernate configuration?

A) The most common methods of Hibernate configuration are:
* Programmatic configuration
* XML configuration (hibernate.cfg.xml)

Q27. What are the important tags of hibernate.cfg.xml?
A)

```
 <hibernate-configuration>
        <session-factory name=" ">
        <property name="  ">  </property>

        <!-- mapping files -->
        <mapping resource=" "/>
        <mapping resource=" "/>

        <!-- cache settings -->
        <class-cache class="org.hibernate.auction.Item" usage="read-write"/>
        <class-cache class="org.hibernate.auction.Bid" usage="read-only"/>
        <collection-cache collection="org.hibernate.auction.Item.bids" usage="read-write"/>
```

Q28. What are the Core interfaces are of Hibernate framework?
A) People who read this also read:
The five core interfaces are used in just about every Hibernate application. Using these interfaces, you can store and retrieve persistent objects and control transactions.

* Session interface
* SessionFactory interface
* Configuration interface
* Transaction interface
* Query and Criteria interfaces

Q29. What role does the Session interface play in Hibernate?
A) The Session interface is the primary interface used by Hibernate applications. It is a single-threaded, short-lived object representing a conversation between the application and the persistent store. It allows you to create query objects to retrieve persistent objects.

```
Session session = sessionFactory.openSession();
```

Session interface role:
* Wraps a JDBC connection
* Factory for Transaction
* Holds a mandatory (first-level) cache of persistent objects, used when navigating the object graph or looking up objects by identifier

Q30. What role does the SessionFactory interface play in Hibernate?
A) The application obtains Session instances from a SessionFactory. There is typically a single SessionFactory for the whole application—created during application initialization. The SessionFactory caches generate SQL statements and other mapping metadata that Hibernate uses at runtime. It also holds cached data that has been read in one unit of work and may be reused in a future unit of work

```
SessionFactory sessionFactory = configuration.buildSessionFactory();
```

Q31. What is Hibernate Query Language (HQL)?
A) Hibernate offers a query language that embodies a very powerful and flexible mechanism to query, store, update, and retrieve objects from a database. This language, the Hibernate query Language (HQL), is an object-oriented extension to SQL.

Q32. What is the general flow of Hibernate communication with RDBMS?
A) The general flow of Hibernate communication with RDBMS is :
* Load the Hibernate configuration file and create configuration object. It will automatically load all hbm mapping files
* Create session factory from configuration object
* Get one session from this session factory
* Create HQL Query
* Execute query to get list containing Java objects

Q33. How do you map Java Objects with Database tables?
A)
* First we need to write Java domain objects (beans with setter and getter). The variables should be same as database columns.
* Write hbm.xml, where we map java class to table and database columns to Java class variables.

Example :
```
<hibernate-mapping>
<class name="com.test.User"  table="user">
<property  column="USER_NAME" length="255″
name="userName" not-null="true"  type="java.lang.String"/>
<property  column="USER_PASSWORD" length="255″
name="userPassword" not-null="true"  type="java.lang.String"/>
</class>
</hibernate-mapping>
```

Q34. What Does Hibernate Simplify?
A) Hibernate simplifies:
* Saving and retrieving your domain objects
* Making database column and table name changes
* Centralizing pre save and post retrieve logic
* Complex joins for retrieving related items
* Schema creation from object model

Q35. What's the difference between load() and get()?
A) load() vs. get()

load() :-
-If you are sure that the object exists then only use the load() method.
-load() method will throw an exception 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():-
-If you are not sure that the object exists, then use one of the get() methods.
-get() method will return null if the unique id is not found in the database.
-get() will hit the database immediately.

Q36. What is the difference between and merge and update ?
A)Use update() if you are sure that the session does not contain an already persistent instance with the same identifier, and merge() if you want to merge your modifications at any time without consideration of the state of the session.

Q37. What does it mean to be inverse?
A) It informs hibernate to ignore that end of the relationship. If the one–to–many was marked as inverse, hibernate would create a child–>parent relationship (child.getParent). If the one–to–many was marked as non–inverse then a child–>parent relationship would be created.

Q38. How do you define sequence generated primary key in hibernate?
A) Using <generator> tag.

Example:-
```
<id column=”USER_ID” name=”id” type=”java.lang.Long”>
<generator class=”sequence”>
<param name=”table”>SEQUENCE_NAME</param>
<generator>
</id>
```

Q39. Define cascade and inverse option in one-many mapping?
A) cascade – enable operations to cascade to child entities.

cascade=”all|none|save-update|delete|all-delete-orphan”

inverse – mark this collection as the “inverse” end of a bidirectional association.

inverse=”true|false”

Essentially “inverse” indicates which end of a relationship should be ignored, so when persisting a parent who has a collection of children, should you ask the parent for its list of children, or ask the children who the parents are?

Q40. What do you mean by Named – SQL query?
A) Named SQL queries are defined in the mapping xml document and called wherever required.
Example:
```
<sql-query name = “empdetails”>
   <return alias=”emp” class=”com.test.Employee”/>
        SELECT emp.EMP_ID AS {emp.empid},
                  emp.EMP_ADDRESS AS {emp.address},
                  emp.EMP_NAME AS {emp.name}
         FROM Employee EMP WHERE emp.NAME LIKE :name
</sql-query>
```

<u>Invoke Named Query :</u>
```
List people = session.getNamedQuery(“empdetails”).setString(“TomBrady”, name)
.setMaxResults(50).list();
```

Q41. How do you invoke Stored Procedures?
A)
```
<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>
```

Q42. Explain Criteria API
A) 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.

Example :
```
List employees = session.createCriteria(Employee.class)
                              .add(Restrictions.like(“name”, “a%”) )
                              .add(Restrictions.like(“address”, “Boston”)).addOrder(Order.asc(“name”) ).list();
```

Q43.If you want to see the Hibernate generated SQL statements on console, what should we do?
A) In Hibernate configuration file set as follows:
<property name=”show_sql”>true</property>

Q44. Define HibernateTemplate?
A) org.springframework.orm.hibernate.HibernateTemplate is a helper class which provides different methods for querying/retrieving data from the database. It also converts checked HibernateExceptions into unchecked DataAccessExceptions.

Q45. What are the benefits does HibernateTemplate provide?
A) The benefits of HibernateTemplate are :
* HibernateTemplate, a Spring Template class simplifies interactions with Hibernate Session.
* Common functions are simplified to single method calls.
* Sessions are automatically closed.
* Exceptions are automatically caught and converted to runtime exceptions.

Q46. How do you switch between relational databases without code changes?
A) Using Hibernate SQL Dialects , we can switch databases. Hibernate will generate appropriate hql queries based on the dialect defined.

Q47.What are derived properties?
A) The properties that are not mapped to a column, but calculated at runtime by evaluation of an expression are called derived properties. The expression can be defined using the formula attribute of the element.

Q48. What is component mapping in Hibernate?
A)
* A component is an object saved as a value, not as a reference
* A component can be saved directly without needing to declare interfaces or identifier properties
* Required to define an empty constructor
* Shared references not supported

Q49. What is the difference between sorted and ordered collection in hibernate?
A) sorted collection vs. order collection

<u>sorted collection :-</u>
A sorted collection is sorting a collection by utilizing the sorting features provided by the Java collections framework. The sorting occurs in the memory of JVM which running Hibernate, after the data being read from database using java comparator.
If your collection is not large, it will be more efficient way to sort it.

<u>order collection :-</u>
Order collection is sorting a collection by specifying the order-by clause for sorting this collection when retrieval.
If your collection is very large, it will be more efficient way to sort it .

***Hibernate* interview questions**

Q1.Difference between session.save() , session.saveOrUpdate() and session.persist()?

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 :
if you do :-

```
System.out.println(session.save(question));
```

This will print the generated primary key.

if you do :-

```
System.out.println(session.persist(question));
```

Compile time error because session.persist() return void.

Q2. What is lazy fetching in Hibernate? With Example .
Lazy fetching decides whether to load child objects while loading the Parent Object.
You need to do this setting respective hibernate mapping file of the parent class.
Lazy = true (means not to load child)

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.

Example :
If you have a TABLE ? EMPLOYEE mapped to Employee object and contains set of Address objects.

Parent Class : Employee class
Child class : Address Class

```
public class Employee {
private Set address = new HashSet(); // contains set of child Address objects
public Set getAddress () {
return address;
}
public void setAddresss(Set address) {
this. address = address;
}
}
```

In the Employee.hbm.xml file

```
<set name="address" inverse="true" cascade="delete" lazy="false">
<key column="a_id" />
<one-to-many class="beans Address"/>
```

```
</set>
```

In the above configuration.

If lazy="false" : - when you load the Employee object that time child object Adress is also loaded and set to setAddresss() method.
If you call employee.getAdress() then loaded data returns.No fresh database call.

If lazy="true" :- This the default configuration. If you don?t mention then hibernate consider lazy=true. when you load the Employee object that time child object Adress is not loaded. You need extra call to data base to get address objects.
If you call employee.getAdress() then that time database query fires and return results. Fresh database call

Q3.How to Integrate Struts ,Spring & Hibernate ?
Details with code you can deploy in tomcat server and test .

Step 1. In the struts-config.xml,add plugin

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
<set-property  property="contextConfigLocation" value="/WEB-INF/applicationContext.xml"/>
</plug-in>
```

Step 2. In the applicationContext.xml file Configure, datasourse

```
<bean id="dataSource"
                    class="org.springframework.jdbc.datasource.DriverManagerDataSource ">
         <property name="driverClassName">
           <value>oracle.jdbc.driver.OracleDriver</value>
         </property>

         <property name="url">
            <value>jdbc:oracle:thin:@10.10.01.24:1541:ebizd</value>
         </property>

         <property name="username">
             <value>sa</value>
         </property>

        <property name="password">
          <value></value>
        </property>
</bean>
```

Step 3. Configure SessionFactory

```
<!-- Hibernate SessionFactory -->
<bean id="sessionFactory"   class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
   <property name="dataSource">
         <ref local="dataSource"/>
   </property>

   <property name="mappingResources">
       <list>
          <value>com/test/dbxml/User.hbm.xml</value>
       </list>
  </property>

<property name="hibernateProperties">
  <props>
```

```
     <prop key="hibernate.dialect">net.sf.hibernate.dialect.OracleDialect </prop>
   </props>
</property>
</bean>
```

Step 4. Configure User.hbm.xml

```
<hibernate-mapping>
<class name="org.test.model.User" table="app_user">

    <id name="id" column="id" >
      <generator class="increment"/>
   </id>
  <property name="firstName" column="first_name" not-null="true"/>
  <property name="lastName" column="last_name" not-null="true"/>

</class>
</hibernate-mapping>
```

Step 5. In the applicationContext.xml ? configure for DAO

```
<bean id="userDAO" class="org.test.dao.hibernate.UserDAOHibernate">
   <property name="sessionFactory">
         <ref local="sessionFactory"/>
   </property>
</bean>
```

Step 6. DAO Class

```
public class UserDAOHibernate extends HibernateDaoSupport implements UserDAO {

    private static Log log = LogFactory.getLog(UserDAOHibernate.class);

    public List getUsers() {
       return getHibernateTemplate().find("from User");
    }

   public User getUser(Long id) {
     return (User) getHibernateTemplate().get(User.class, id);
   }

   public void saveUser(User user) {
     getHibernateTemplate().saveOrUpdate(user);

    if (log.isDebugEnabled()) {
     log.debug("userId set to: " + user.getId());
    }
  }

 public void removeUser(Long id) {
       Object user = getHibernateTemplate().load(User.class, id);
    getHibernateTemplate().delete(user);
 }
}
```

Q4.How to prevent concurrent update in Hibernate? (OR) How to perevent slate object updation in Hibernate ? (OR) What is version checking in Hibernate ? (OR)
How to handle user think time using hibernate ?

Version checking used in hibernate when more then one thread trying to access same data.

For example : User A edit the row of the TABLE for update (In the UI changing data - This is user thinking time) and in the same time User B edit the same record for update and click the update. Then User A click the Update and update done. Chnage made by user B is gone.

In hibernate you can perevent slate object updatation using version checking.

Check the version of the row when you are upding the row.Get the version of the row when you are fetching the row of the TABLE for update.On the time of updation just fetch the version number and match with your version number (on the time of fetching).

This way you can prevent slate object updation.

Steps 1:
Declare a variable "versionId" in your Class with setter and getter.

```
public class Campign {
private Long versionId;
private Long campignId;
private String name;
public Long getVersionId() {
        return versionId;
        }
public void setVersionId(Long versionId) {
        this.versionId = versionId;
        }

public String getName() {
        return name;
        }
public void setName(String name) {
        this.name = name;
        }

public Long getCampignId() {
      return campignId;
   }

private void setCampignId(Long campignId) {
      this.campignId = campignId;
   }
}
```

Step 2.
In the .hbm.xml file

```
<class name="beans.Campign" table="CAMPIGN" optimistic-lock="version">
<id name="campignId" type="long" column="cid">
        <generator class="sequence">
                <param name="sequence">CAMPIGN_ID_SEQ</param>
        </generator>
  </id>
   <version name="versionId" type="long" column="version" />
<property name="name" column="c_name"/>
```

</class>

Step 3.
Create a coulmn name "version" in the CAMPIGN table.

Step 4.
In the code

```
// foo is an instance loaded by a previous Session
session = sf.openSession();
int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() );
if ( oldVersion!=foo.getVersion ) throw new StaleObjectStateException();
foo.setProperty("bar");
session.flush();
session.connection().commit();
session.close();
```

You can handle StaleObjectStateException() and do what ever you want.
You can display error message.
Hibernate autumatically create/update the version number when you update/insert any row in the table.

Q5.Transaction with plain JDBC in Hibernate ?

If you don't have JTA and don't want to deploy it along with your application, you will usually have to fall back to JDBC transaction demarcation. Instead of calling the JDBC API you better use Hibernate's Transaction and the built-in session-per-request functionality:

To enable the thread-bound strategy in your Hibernate configuration:

```
set hibernate.transaction.factory_class to org.hibernate.transaction.JDBCTransactionFactory
set hibernate.current_session_context_class to thread
```

```
Session session = factory.openSession();
Transaction tx = null;
try {
tx = session.beginTransaction();

// Do some work
session.load(...);
session.persist(...);

tx.commit(); // Flush happens automatically
}
catch (RuntimeException e) {
tx.rollback();
throw e; // or display error message
}
finally {
session.close();
}
```

Q6.What are the general considerations or best practices for defining your Hibernate persistent classes?
1.You must have a default no-argument constructor for your persistent classes and there should be getXXX() (i.e accessor/getter) and setXXX(i.e. mutator/setter) methods for all your persistable instance variables.

2.You should implement the equals() and hashCode() methods based on your business key and it is important not to use the id field in your equals() and hashCode() definition if the id field is a surrogate key (i.e. Hibernate managed identifier). This is because the Hibernate only generates and sets the field when saving the object.

3. It is recommended to implement the Serializable interface. This is potentially useful if you want to migrate around a multi-processor cluster.

4.The persistent class should not be final because if it is final then lazy loading cannot be used by creating proxy objects.

Q7.Difference between session.update() and session.lock() in Hibernate ?

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 "reassociates" 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.

Q8.Difference between getCurrentSession() and openSession() in Hibernate ?
<u>getCurrentSession() :</u>

A Session is opened when getCurrentSession() is called for the first time and closed when the transaction ends. It is also flushed automatically before the transaction commits. You can call getCurrentSession() as often and anywhere you want as long as the transaction runs.

<u>To enable this strategy in your Hibernate configuration:</u>
set hibernate.transaction.manager_lookup_class to a lookup strategy for your JEE container
set hibernate.transaction.factory_class to org.hibernate.transaction.JTATransactionFactory

Only the Session that you obtained with sf.getCurrentSession() is flushed and closed automatically.

Example :
```
try {
        UserTransaction tx = (UserTransaction)new InitialContext().lookup("java:comp/UserTransaction");

        tx.begin();

          // Do some work
          sf.getCurrentSession().createQuery(...);
          sf.getCurrentSession().persist(...);

          tx.commit();

    } catch (RuntimeException e) {
          tx.rollback();
          throw e; // or display error message
     }
```

<u>openSession() :</u>
If you decide to use and manage the Session yourself the go for sf.openSession() , you have to flush() and close() it. It does not flush and close() automatically.

Example :
```
 UserTransaction tx = (UserTransaction)new InitialContext() .lookup("java:comp/UserTransaction");

Session session = factory.openSession();
try {
          tx.begin();

          // Do some work
          session.createQuery(...);
          session.persist(...);

          session.flush(); // Extra work you need to do

          tx.commit();
   } catch (RuntimeException e) {
          tx.rollback();
          throw e; // or display error message
  } finally {
          session.close(); // Extra work you need to do
}
```

Q9.Difference between session.saveOrUpdate() and session.merge()?
<u>saveOrUpdate() does the following:</u>
- if the object is already persistent in this session, do nothing
- if another object associated with the session has the same identifier, throw an exception
- if the object has no identifier property, save() it
- if the object's identifier has the value assigned to a newly instantiated object, save() it
- 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 otherwise update() the object

<u>merge()is very different:</u>
- 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 .
- if there is no persistent instance currently associated with the session, try to load it from the database, or create a new persistent instance
- the persistent instance is returned
- the given instance does not become associated with the session, it remains detached

Q10.Filter in Hibernate with Example?

Filter in Hibernate ------

USER (ID INT, USERNAME VARCHAR, ACTIVATED BOOLEAN) - TABLE

```
public class User
{
        private int id;
        private String username;
        private boolean activated;
        public boolean isActivated()
        {
         return activated;
        }
        public void setActivated(boolean activated)
        {
        this.activated = activated;
        }
        public int getId()
        {
        return id;
        }
        public void setId(int id)
        {
        this.id = id;
        }
        public String getUsername()
        {
        return username;
        }
public void setUsername(String username)
        {
        this.username = username;
        }
}
```

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
<class name="User">
        <id name="id" type="int">
         <generator class="native"/>
        </id>
        <property name="username" type="string" length="32"/>
        <property name="activated" type="boolean"/>
        <filter name="activatedFilter" condition=":activatedParam = activated"/>
</class>

<filter-def name="activatedFilter">
<filter-param name="activatedParam" type="boolean"/>
</filter-def>

</hibernate-mapping>
```

Save and Fetch using filter example

```
User user1 = new User();
user1.setUsername("name1");
user1.setActivated(false);
session.save(user1);

User user2 = new User();
user2.setUsername("name2");
user2.setActivated(true);
session.save(user2);

User user3 = new User();
user3.setUsername("name3");
user3.setActivated(true);
session.save(user3);

User user4 = new User();
user4.setUsername("name4");
user4.setActivated(false);
session.save(user4);
```

All the four user saved to Data Base User Table.
Now Fetch the User using Filter..

```
Filter filter = session.enableFilter("activatedFilter");
filter.setParameter("activatedParam",new Boolean(true));
Query query = session.createQuery("from User");
Iterator results = query.iterate();
while (results.hasNext())
{
User user = (User) results.next();
System.out.print(user.getUsername() + " is ");
}
```

Guess the Result :
name2 name3

Because Filer is filtering (only true value) data before query execute.

Q11.How does Value replacement in Message Resource Bundle work?
In the resource bundle file, you can define a template like:

```
errors.required={0} is required.
ActionErrors errors = new ActionErrors();
errors.add(ActionErrors.GLOBAL_ERROR,
new ActionError("error.custform","First Name"));
```

Then the Error message is : First Name is required.
Other constructors are

```
public ActionError(String key, Object value0, Object value1)
. . .
public ActionError(String key, Object[] values);
```

Q12.Difference between list() and iterate() i9n Hibernate?
If instances are already be in the session or second-level cache iterate() will give better performance. If they are not already cached, iterate() will be slower than list() and might require many database hits for a simple query.

Q13.Difference between session.load() and session.get() ?
load() will throw an unrecoverable exception if there is no matching database row.
get() will return null if there is no matching database row.

```
Cat fritz = (Cat) session.load(Cat.class, "1");
```

Return the Cat Object with key 1. If there is no Cat Object with key 1 then throw will throw an unrecoverable exception.

If the class is mapped with a proxy, load() just returns an uninitialized proxy and does not actually hit the database until you invoke a method of the proxy. This behaviour is very useful if you wish to create an association to an object without actually loading it from the database. It also allows multiple instances to be loaded as a batch if batchsize is defined for the class mapping.

```
Cat fritz = (Cat) session.get(Cat.class, "1");
```

If you are not certain that a matching row exists, you should use the get() method, which hits the database immediately and returns null if there is no matching row.

Q14.Deleting persistent objects
Session.delete() will remove an object's state from the database. Of course, your application might still hold a reference to a deleted object. It's best to think of delete() as making a persistent instance transient.

```
session.delete(cat);
```

Q15.SQL statements execution order.
The SQL statements are issued in the following order
1. all entity insertions, in the same order the corresponding objects were saved using Session.save()
2. all entity updates
3. all collection deletions
4. all collection element deletions, updates and insertions
5. all collection insertions
6. all entity deletions, in the same order the corresponding objects were deleted using Session.delete()
(An exception is that objects using native ID generation are inserted when they are saved.)

Except when you explicity flush(), there are absolutely no guarantees about *when* the Session executes the JDBC calls, only the *order* in which they are executed. However, Hibernate does guarantee that the Query.list(..) will never return stale data; nor will they return the wrong data

Q16.Criteria Query Two Condition
Criteria Query Two Condition- Example

```
<class name="com.bean.Organization" table="ORGANIZATION">
        <id name="orgId" column="ORG_ID" type="long">
        <generator class="native"/>
        </id>
        <property name="organizationName" column="ORGANISATION_NAME" type="string"
length="500"/>
        <property name="town" column="TOWN" type="string" length="200"/>
        <property name="statusCode" column="STATUS" type="string" length="1"/>
</class>
```

List of organisation where town equals to pune and status = "A".

```
List organizationList = session.createCriteria(Organization.class)
.add(Restrictions.eq("town","pune")).add(Restrictions.eq("statusCode","A")).list();
```

Q17.Equal and Not Equal criteria query.

```
Equal and Not Equal criteria query- Example
<class name="com.bean.Organization" table="ORGANIZATION">
<id name="orgId" column="ORG_ID" type="long">
<generator class="native"/>
</id>
<property name="organizationName" column="ORGANISATION_NAME" type="string" length="500"/>
<property name="town" column="TOWN" type="string" length="200"/>
</class>
List of organisation where town equals to pune.

List organizationList =
session.createCriteria(Organization.class).add(Restrictions.eq("town","pune")).list();

List of organisation where town not equals pune.
List organizationList =
session.createCriteria(Organization.class).add(Restrictions.ne("town","pune")).list();
```

Q18.Cascade Save or Update in Hibernate ?

Cascade Save or Update - In one to Many- EXAMPLE

PROCESS_TYPE_LOV (PROCESS_TYPE_ID number, PROCESS_TYPE_NAME varchar) - TABLE
PROCESS (PROCESS_ID number,PROCESS_NAME varchar,PROCESS_TYPE_ID number)- TABLE

```
public class ProcessTypeBean {

    private Long processTypeId;
    private String processTypeName;

    public Long getProcessTypeId() {
        return processTypeId;
    }

    public void setProcessTypeId(Long processTypeId) {
        this.processTypeId = processTypeId;
    }

    public String getProcessTypeName() {
        return processTypeName;
    }

    public void setProcessTypeName(String processTypeName) {
        this.processTypeName = processTypeName;
    }

}

public class ProcessBean {
    private Long processId;
    private String processName = "";
    private ProcessTypeBean processType;

    public Long getProcessId() {
        return processId;
    }
    public void setProcessId(Long processId) {
        this.processId = processId;
    }

    public String getProcessName() {
        return processName;
    }

    public void setProcessName(String processName) {
        this.processName = processName;
    }

    public ProcessTypeBean getProcessType() {
        return processType;
    }

    public void setProcessType(ProcessTypeBean processType) {
        this.processType = processType;
    }
}

<class name="com.bean.ProcessBean" table="PROCESS">
        <id name="processId" type="long" column="PROCESS_ID" />
        <property name="processName" column="PROCESS_NAME" type="string"  length="50" />
        <many-to-one name="processType" column="PROCESS_TYPE_ID" class="ProcessTypeBean"
cascade="save-update" />
</class>

<class name="com.bean.ProcessTypeBean" table="PROCESS_TYPE_LOV">
        <id name="processTypeId" type="long" column="PROCESS_TYPE_ID" />
        <property name="processTypeName" column="PROCESS_TYPE_NAME"
            type="string" length="50" />
    </class>
```

Save Example Code -

```
ProcessTypeBean pstype = new ProcessTypeBean();
pstype.setProcessTypeName("Java Process");

ProcessBean process = new ProcessBean();
process.setProcessName("Production")
ProcessBean.setProcessType(pstype);

// session.save(pstype); -- This save not required because of in the mapping file cascade="save-update"
session.save(process); - This will insert both ProcessBean and ProcessTypeBean;
```

Q19.One To Many Bi-directional Relation in Hibernate?
PROCESS_TYPE_LOV (PROCESS_TYPE_ID number, PROCESS_TYPE_NAME varchar) - TABLE
PROCESS (PROCESS_ID number,PROCESS_NAME varchar,PROCESS_TYPE_ID number)- TABLE

```
public class ProcessTypeBean {

    private Long processTypeId;
    private String processTypeName;
    private List processes = null;

        public List getProcesses() {
        return processes;
    }

    public void setProcesses(List processes) {
        this.processes = processes;
    }

public Long getProcessTypeId() {
        return processTypeId;
    }

    public void setProcessTypeId(Long processTypeId) {
        this.processTypeId = processTypeId;
    }

    public String getProcessTypeName() {
        return processTypeName;
    }

    public void setProcessTypeName(String processTypeName) {
        this.processTypeName = processTypeName;
    }
}

public class ProcessBean {
    private Long processId;
    private String processName = "";
    private ProcessTypeBean processType;

    public Long getProcessId() {
        return processId;
    }

    public void setProcessId(Long processId) {
        this.processId = processId;
    }

    public String getProcessName() {
        return processName;
    }

    public void setProcessName(String processName) {
        this.processName = processName;
    }

    public ProcessTypeBean getProcessType() {
        return processType;
    }

    public void setProcessType(ProcessTypeBean processType) {
        this.processType = processType;
    }
}

<class name="com.bean.ProcessBean"
        table="PROCESS">
        <id name="processId" type="long" column="PROCESS_ID" />
        <property name="processName" column="PROCESS_NAME" type="string"  length="50" />
        <many-to-one name="processType" column="PROCESS_TYPE_ID" lazy="false" />
</class>

<class name="com.bean.ProcessTypeBean" table="PROCESS_TYPE_LOV">
        <id name="processTypeId" type="long" column="PROCESS_TYPE_ID" />
        <property name="processTypeName" column="PROCESS_TYPE_NAME"
            type="string" length="50" />

        <bag name="processes" inverse="true" cascade="delete" lazy="false">
            <key column="PROCESS_TYPE_ID" />
            <one-to-many
                class="com.bean.ProcessBean" />
        </bag>

    </class>
```

Q20.One To Many Mapping Using List ?
WRITER (ID INT,NAME VARCHAR) - TABLE
STORY (ID INT,INFO VARCHAR,PARENT_ID INT) - TABLE

One writer can have multiple stories.. Mapping File…

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
<class name="Writer" table="WRITER">
<id name="id" unsaved-value="0">
<generator class="increment"/>
</id>

<list name="stories" cascade="all">
          <key column="parent_id"/>
          <one-to-many class="Story"/>
</list>
<property name="name" type="string"/>
</class>
<class name="Story" table="story">
<id name="id" unsaved-value="0">
<generator class="increment"/>
</id>
<property name="info"/>
</class>
</hibernate-mapping>
```

```
public class Writer {
private int id;
private String name;
private List stories;

public void setId(int i) {
id = i;
}

public int getId() {
return id;
}

public void setName(String n) {
name = n;
}
public String getName() {
return name;
}

public void setStories(List l) {
stories = l;
}

public List getStories() {
return stories;
}
}
---------------------------------------------------
public class Story {
private int id;
private String info;

public Story(){
}

public Story(String info) {
this.info = info;
}

public void setId(int i) {
id = i;
}

public int getId() {
return id;
}

public void setInfo(String n) {
info = n;
}

public String getInfo() {
return info;
}
}
```
-------------------------Save Example ..

```
Writer wr = new Writer();
wr.setName("Das");

ArrayList list = new ArrayList();
list.add(new Story("Story Name 1"));
list.add(new Story("Story Name 2"));
wr.setStories(list);

Transaction transaction = null;

try {
        transaction = session.beginTransaction();
        session.save(sp);
        transaction.commit();
} catch (Exception e) {
        if (transaction != null) {
                transaction.rollback();
                throw e;
        }
} finally {
session.close();
}
```

Q21.Many To Many Relation In Hibernate ?
EVENTS (uid int, name VARCHAR) Table
SPEAKERS (uid int, firstName VARCHAR) Table
EVENT_SPEAKERS (elt int, event_id int, speaker_id int) Table

```
-----------------------------------------------------------
import java.util.Set;
import java.util.HashSet;

public class Speaker{
private Long id;
private String firstName;
private Set events;

public Long getId() {
return id;
}

public void setId(Long id) {
this.id = id;
}
public String getFirstName() {
return firstName;
}

public void setFirstName(String firstName) {
this.firstName = firstName;
}

public Set getEvents() {
return this.events;
}

public void setEvents(Set events) {
this.events = events;
}
private void addEvent(Event event) {
if (events == null) {
events = new HashSet();
}
events.add(event);
}
}
-----------------------------------------------------
import java.util.Date;
import java.util.Set;

public class Event{
private Long id;
private String name;
private Set speakers;

public void setId(Long id) {
this.id = id;
}

public Long getId() {
return id;
}

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}

public void setSpeakers(Set speakers) {
this.speakers = speakers;
}

public Set getSpeakers() {
return speakers;
}
}
-----------------------------------------------------------
```

Event.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
<class name="Event" table="events">
<id name="id" column="uid" type="long" unsaved-value="null">
<generator class="increment"/>
</id>
<property name="name" type="string" length="100"/>
<set name="speakers" table="event_speakers" cascade="all">
<key column="event_id"/>
<many-to-many class="Speaker"/>
</set>
</class>
```

```
</hibernate-mapping>
```

Speaker.hbm.xml

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 2.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-2.0.dtd">

<hibernate-mapping>
<class name="Speaker" table="speakers">
<id name="id" column="uid" type="long">
<generator class="increment"/>
</id>
<property name="firstName" type="string" length="20"/>
        <set name="events" table="event_speakers" cascade="all">
        <key column="speaker_id"/>
        <many-to-many class="Event"/>
        </set>
</class>
</hibernate-mapping>
```

Save and Fetch Example

```
Event event = new Event();
event.setName("Inverse test");
event.setSpeakers(new HashSet());
event.getSpeakers().add(new Speaker("Ram", event));
event.getSpeakers().add(new SpeakerManyToMany("Syam", event));
event.getSpeakers().add(new SpeakerManyToMany("Jadu", event));
session.save(event); /// Save All the Data

event = (Event) session.load(Event.class, event.getId());
Set speakers = event.getSpeakers();

for (Iterator i = speakers.iterator(); i.hasNext();) {
Speaker speaker = (Speaker) i.next();
System.out.println(speaker.getFirstName());
System.out.println(speaker.getId());
}
```

Q22.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)
```

Q23.Difference between session.load() and session.get()?

```
Cat fritz = (Cat) session.load(Cat.class, "1");
```

Return the Cat Object with key 1. If there is no Cat Object with key 1 then throw will throw an unrecoverable exception.

If the class is mapped with a proxy, load() just returns an uninitialized proxy and does not actually hit the database until you invoke a method of the proxy. This behaviour is very useful if you wish to create an association to an object without actually loading it from the database. It also allows multiple instances to be loaded as a batch if batchsize is defined for the class mapping.

```
Cat fritz = (Cat) session.get(Cat.class, "1");
```

If you are not certain that a matching row exists, you should use the get() method, which hits the database immediately and returns null if there is no matching row.

Q24.How to add .hbm.xml file in sessionFactory?

```
 SessionFactory sf = new Configuration()
.addFile("Item.hbm.xml")
.addFile("Bid.hbm.xml")
.buildSessionFactory();
```

Q25.How to get Hibernate statistics ?

If you enable hibernate.generate_statistics, Hibernate will expose a number of metrics that are useful when tuning a running system via SessionFactory.getStatistics().

SessionFactory.getStatistics() is give you all the statistics .

Q26.How to get JDBC connections in hibernate?

Use Session.connection() method to get JDBC Connection

Q27.Hibernate setup using .cfg.xml file ?

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

```
<?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  name="java:hibernate/SessionFactory">
 <property name="connection.datasource">java:/comp/env/jdbc/MyDB</property>
 <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
 <property name="show_sql">false</property>
<property name="transaction.factory_class"> org.hibernate.transaction.JTATransactionFactory
</property>
<property name="jta.UserTransaction">java:comp/UserTransaction</property>
<mapping resource="org/hibernate/auction/Cost.hbm.xml"/>
</session-factory>
</hibernate-configuration>
```

As you can see, the advantage of this approach is the externalization of the mapping file names to configuration. The hibernate.cfg.xml is also more convenient once you have to tune the Hibernate cache.

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

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

You can pick a different XML configuration file using

```
SessionFactory sf = new Configuration().configure("catdb.cfg.xml").buildSessionFactory();
```

Cost.hbm.xml -----> looks like

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
"-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
<class name="com.bean.Cost" table="COST">
<id name="id" column="ID">
</id>
<property name="isQueued" type="int" column="IS_QUEUED"/>
<property name="queueDate" type="timestamp" column="QUEUE_DATE"/>
<property name="lastModifiedDate" type="timestamp" column="LAST_MODIFIED_DATE"/>
<property name="lastModifiedBy" column="LAST_MODIFIED_BY"/>
<property name="amount" column="AMOUNT" type="double"/>
<property name="currencyCode" column="CURRENCY_CODE" />
<property name="year" column="YEAR"/>
<property name="quarter" column="QUARTER"/>
<property name="costModFlag" type="int" column="COST_MOD_FLAG"/>
<property name="parentId" column="PARENT_ID"/>
<property name="oldParentId" column="OLD_PARENT_ID"/>
<property name="parentIdModFlag" type="int" column="PARENT_ID_MOD_FLAG"/>
<property name="dateIncurred" type="timestamp" column="DATE_INCURRED"/>
<property name="USDAmount" column="USD_AMOUNT" type="double"/>
<property name="isDeleted" type="int" column="IS_DELETED"/>
</class>

</hibernate-mapping>
```

Q28.How to set 2nd level cache in hibernate with EHCache?
When you are creating SessionFactory just add the below steps

```
String ecache = appHome+File.separatorChar+"ehcache.xml";
 try {
         CacheManager.create(ecache);
     } catch (CacheException e) {
         // logger.logError(e);
    }*/
```

Then, sessionFactory = configuration.buildSessionFactory();

ECache.xml is like

```
<ehcache>
    <diskStore path="java.io.tmpdir"/>
      <defaultCache  maxElementsInMemory="10000"
                            eternal="false"
                            timeToIdleSeconds="120"
                            timeToLiveSeconds="120"
                            overflowToDisk="true"
                           diskPersistent="false"
                          diskExpiryThreadIntervalSeconds="120"/>

    <cache name="bean.ApplicationBean" maxElementsInMemory="300" eternal="false"
                  overflowToDisk="false" />
</ehcache>
```

ApplicationBean will be avilable in 2nd level cache

Q29.How will you configure Hibernate?
Step 1> Put Hibernate properties in the classpath.

Step 2> Put .hbm.xml in class path ?
Code is Here to create session ...

```
package com.dao;

import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;

import org.apache.log4j.Logger;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    protected static final Logger logger=Logger.getLogger(HibernateUtil.class);
    public static String appHome = "No";

    private static SessionFactory sessionFactory;

    private static final ThreadLocal threadSession = new ThreadLocal();
    private static final ThreadLocal threadTransaction = new ThreadLocal();

    public static void initMonitor(){
        logger.info("Hibernate configure");
        try {
            logger.info("appHome"+appHome);
            String path_properties = appHome+File.separatorChar+"hibernate.properties";
            String path_mapping = appHome+File.separatorChar+"mapping_classes.mysql.hbm.xml";
            //String ecache = appHome+File.separatorChar+"ehcache.xml";

            Properties propHibernate = new Properties();
            propHibernate.load(new FileInputStream(path_properties));

            Configuration configuration = new Configuration();
            configuration.addFile(path_mapping);
            configuration.setProperties(propHibernate);

            /* try {
CacheManager.create(ecache);
} catch (CacheException e) {
// logger.logError(e);
}*/
            sessionFactory = configuration.buildSessionFactory();

        } catch (Throwable ex) {
            logger.error("Exception in initMonitor",ex);
            throw new ExceptionInInitializerError(ex);
        }
```

```
}

public static SessionFactory getSessionFactory() {
    logger.info("Inside getSessionFactory method");
    try {

        if (sessionFactory == null) {
            initMonitor();
        }else {

            //sessionFactory.getStatistics().logSummary();
        }

    } catch (Exception e) {
        logger.error("Exception in getSessionFactory",e);
    }
    return sessionFactory;
}

public static Session getSession() {
    Session s = (Session) threadSession.get();
    logger.debug("session"+s);
    if (s == null) {

        s = getSessionFactory().openSession();
        threadSession.set(s);
        logger.debug("session 1 $"+s);
    }
    return s;
}

public static void closeSession(){
    Session s = (Session) threadSession.get();
    threadSession.set(null);
    if (s != null && s.isOpen()) {
        s.flush();
        s.close();
    }
}

   public static void beginTransaction(){
    Transaction tx = null;
       if (tx == null) {
        tx = getSession().beginTransaction();
        threadTransaction.set(tx);
    }
}

public static void commitTransaction(){
    Transaction tx = (Transaction) threadTransaction.get();
    try {
        if ( tx != null ) {
            tx.commit();
        }
        threadTransaction.set(null);

    } catch (HibernateException ex) {
        rollbackTransaction();
```

```
            throw ex;
        }
    }
    public static void rollbackTransaction(){

        Transaction tx = (Transaction) threadTransaction.get();
        try {
            threadTransaction.set(null);
            if ( tx != null && !tx.wasCommitted() && !tx.wasRolledBack() ) {
                tx.rollback();
            }
        } finally {
            closeSession();
        }
    }

}
```

Q30.How to create Session and SessionFactory in Hibernate?

Step 1> Put Hibernate properties in the classpath.
Step 2> Put .hbm.xml in class path ?

Code is Here to create session ...

```
package com.dao;
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
import org.apache.log4j.Logger;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class HibernateUtil {
    protected static final Logger logger=Logger.getLogger(HibernateUtil.class);
    public static String appHome = "No";

    private static SessionFactory sessionFactory;

    private static final ThreadLocal threadSession = new ThreadLocal();
    private static final ThreadLocal threadTransaction = new ThreadLocal();

    public static void initMonitor(){
        logger.info("Hibernate configure");
        try {
            logger.info("appHome"+appHome);
            String path_properties = appHome+File.separatorChar+"hibernate.properties";
            String path_mapping = appHome+File.separatorChar+"mapping_classes.mysql.hbm.xml";
            //String ecache = appHome+File.separatorChar+"ehcache.xml";

            Properties propHibernate = new Properties();
            propHibernate.load(new FileInputStream(path_properties));
            Configuration configuration = new Configuration();
            configuration.addFile(path_mapping);
            configuration.setProperties(propHibernate);
            /* try {
CacheManager.create(ecache);
} catch (CacheException e) {
// logger.logError(e);
}*/
             sessionFactory = configuration.buildSessionFactory();

        } catch (Throwable ex) {
            logger.error("Exception in initMonitor",ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        logger.info("Inside getSessionFactory method");
        try {
            if (sessionFactory == null) {
                initMonitor();
            }else {
                //sessionFactory.getStatistics().logSummary();
            }
        } catch (Exception e) {
            logger.error("Exception in getSessionFactory",e);
        }
        return sessionFactory;
    }

    public static Session getSession() {
        Session s = (Session) threadSession.get();
        logger.debug("session"+s);
        if (s == null) {
            s = getSessionFactory().openSession();
            threadSession.set(s);
            logger.debug("session 1 $"+s);
        }
        return s;
    }

    public static void closeSession(){
        Session s = (Session) threadSession.get();
        threadSession.set(null);
        if (s != null && s.isOpen()) {
            s.flush();
            s.close();
        }
    }

    public static void beginTransaction(){
        Transaction tx = null;
        if (tx == null) {
            tx = getSession().beginTransaction();
            threadTransaction.set(tx);
        }
    }
     public static void commitTransaction(){
        Transaction tx = (Transaction) threadTransaction.get();
        try {
            if ( tx != null ) {
                tx.commit();
            }
            threadTransaction.set(null);
        } catch (HibernateException ex) {
            rollbackTransaction();
            throw ex;
        }
    }

     public static void rollbackTransaction(){
            Transaction tx = (Transaction) threadTransaction.get();
        try {
            threadTransaction.set(null);
            if ( tx != null && !tx.wasCommitted() && !tx.wasRolledBack() ) {
                tx.rollback();
            }
        } finally {
            closeSession();
        }
    }

    }
```

Q31.What are the core components in Hibernate ?
SessionFactory (org.hibernate.SessionFactory)
A threadsafe (immutable) cache of compiled mappings for a single database. A factory for Session and a client of ConnectionProvider. Might hold an optional (second-level) cache of data that is reusable between transactions, at a process- or cluster-level.

Session (org.hibernate.Session) A single-threaded, short-lived object representing a conversation between the application and the persistent store. Wraps a JDBC connection. Factory for Transaction. Holds a mandatory (first-level) cache of persistent objects, used when navigating the object graph or looking up objects by identifier.

Persistent objects and collections
Short-lived, single threaded objects containing persistent state and business function. These might be ordinary JavaBeans/POJOs, the only special thing about them is that they are currently associated with (exactly one) Session. As soon as the Session is closed, they will be detached and free to use in any application layer (e.g. directly as data transfer objects to and from presentation).

Transient and detached objects and collections
Instances of persistent classes that are not currently associated with a Session. They may have been instantiated by the application and not (yet) persisted or they may have been instantiated by a closed Session.

Transaction (org.hibernate.Transaction) (Optional) A single-threaded, short-lived object used by the application to specify atomic units of work.

Abstracts application from underlying JDBC, JTA or CORBA transaction. A Session might span several Transactions in some cases. However, transaction demarcation, either using the underlying API or Transaction, is never optional!

ConnectionProvider (org.hibernate.connection.ConnectionProvider) (Optional)
A factory for (and pool of) JDBC connections. Abstracts application from underlying Datasource or DriverManager. Not exposed to application, but can be extended/implemented by the developer.

TransactionFactory (org.hibernate.TransactionFactory) (Optional) A factory for Transaction instances. Not exposed to the application, but can be extended/ implemented by the developer.

Extension Interfaces
Hibernate offers many optional extension interfaces you can implement to customize the behavior of your persistence layer.

Q32.What is a Hibernate Session? Can you share a session object between different theads?

Session is a light weight and a non-threadsafe object (No, you cannot share it between threads) that represents a single unit-of-work with the database. Sessions are opened by a SessionFactory and then are closed when all work is complete. Session is the primary interface for the persistence service. A session obtains a database connection lazily (i.e. only when required). To avoid creating too many sessions ThreadLocal class can be used as shown below to get the current session no matter how many times you make call to the currentSession() method.

```
public class HibernateUtil {
public static final ThreadLocal local = new ThreadLocal();
public static Session currentSession() throws HibernateException {
Session session = (Session) local.get();
//open a new session if this thread has no session
if(session == null) {
session = sessionFactory.openSession();
local.set(session);
}
return session;
}
}
```

Q33.addScalar() method in hibernate...

```
Double max = (Double) sess.createSQLQuery("select max(cat.weight) as maxWeight from cats
cat").addScalar("maxWeight", Hibernate.DOUBLE);
.uniqueResult();
```

addScalar() method confim that maxWeight is always double type.

This way you don't need to check for it is double or not.

Q34.Hibernate session.close does _not_ call session.flush ?
session.close() don't call session.flush() before closing the session.

This is the session.close() code in hibernate.jar

```
public Connection close() throws HibernateException {
      log.trace( "closing session" );
      if ( isClosed() ) {
         throw new SessionException( "Session was already closed" );
      }

      if ( factory.getStatistics().isStatisticsEnabled() ) {
         factory.getStatisticsImplementor().closeSession();
      }

      try {
         try {
            if ( childSessionsByEntityMode != null ) {
               Iterator childSessions = childSessionsByEntityMode.values().iterator();
               while ( childSessions.hasNext() ) {
                  final SessionImpl child = ( SessionImpl ) childSessions.next();
                  child.close();
               }
            }
         }
         catch( Throwable t ) {
            // just ignore
         }

         if ( rootSession == null ) {
            return jdbcContext.getConnectionManager().close();
         }
         else {
            return null;
         }
      }
      finally {
         setClosed();
         cleanup();
      }
   }
```

Q35.What is the main difference between Entity Beans and Hibernate ?

1)In Entity Bean at a time we can interact with only one data Base. Where as in Hibernate we can able to establishes the connections to more than One Data Base. Only thing we need to write one more configuration file.

2) EJB need container like Weblogic, WebSphare but hibernate don't nned. It can be run on tomcat.

3) Entity Beans does not support OOPS concepts where as Hibernate does.

4) Hibernate supports multi level cacheing, where as Entity Beans doesn't.

5) In Hibernate C3P0 can be used as a connection pool.

6) Hibernate is container independent. EJB not.

Q36.Difference between session.save() and session.saveOrUpdate()?
session.save() - Insert data into the table
session.saveOrUpdate()- Insert data if primary key not exist otherwise update

Q37.How are joins handled using Hibernate.
Best is use Criteria query

Example -
You have parent class

```
public class Organization {
private long orgId;
private List messages;
}
```

Child class
```
public class Message {
   private long messageId;
private Organization organization;
}
```

.hbm.xml file
```
<class name="com.bean.Organization" table="ORGANIZATION">
<bag name="messages" inverse="true" cascade="delete" lazy="false">
          <key column="MSG_ID" />
          <one-to-many class="com.bean.Message" />
      </bag>
</class>

<class name="com.bean.Message" table="MESSAGE">
   <many-to-one name="organization" column="ORG_ID" lazy="false"/>
</class>
```

Get all the messages from message table where organisation id = <any id>

Criteria query is :
```
session.createCriteria(Message.class).createAlias("organization","org").
          add(Restrictions.eq("org.orgId",new Long(orgId)))
.add(Restrictions.in("statusCode",status)).list();
```

Q38.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>
code :
SQLQuery sq = (SQLQuery) session.getNamedQuery("selectAllEmployees_SP");

List results = sq.list();
```

Q39.What is Hibernate proxy?
By default Hibernate creates a proxy for each of the class you map in mapping file. This class contain the code to invoke JDBC. This class is created by hibernate using CGLIB.

Proxies are created dynamically by subclassing your object at runtime. The subclass has all the methods of the parent, and when any of the methods are accessed, the proxy loads up the real object from the DB and calls the method for you. Very nice in simple cases with no object hierarchy. Typecasting and instanceof work perfectly on the proxy in this case since it is a direct subclass.

Q40.what is lazy fetching in hibernate?
Lazy setting decides whether to load child objects while loading the Parent Object.You need to do this setting respective hibernate mapping file of the parent class.Lazy = true (means not to load child)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.Exampleslazy=true (default)Address child of User class can be made lazy if it is not required frequently.lazy=falseBut you may need to load the Author object for Book parent whenever you deal with the book for online bookshop

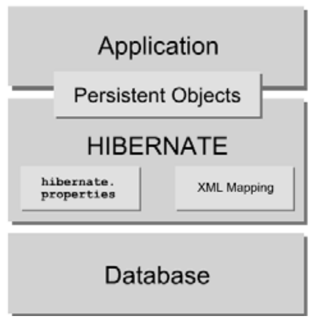
ORM
The term object/relational mapping (ORM) refers to the technique of mapping a data representation from an object model to a relational data model with a SQL-based schema. Object/relational mappings are usually defined in an XML document.

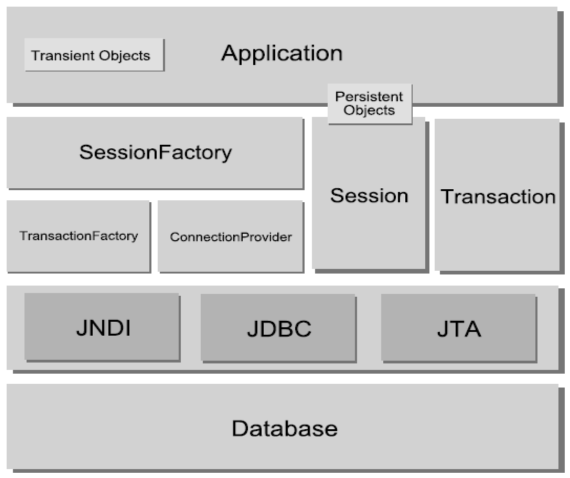
Note:
object/relational mapping is the automated (and transparent) persistence of objects in a Java application to the tables in a relational database, using metadata that describes the mapping between the objects and the database.

An ORM solution consists of the following four pieces:
■ An API for performing basic CRUD operations on objects of persistent classes.
■ A language or API for specifying queries that refer to classes and properties of classes.
■ A facility for specifying mapping metadata.
■ A technique for the ORM implementation to interact with transactional objects to perform dirty checking, lazy association fetching, and other optimization functions.

Hibernate
-Hibernate is an object/relational mapping tool for Java environments.
-Hibernates goal is to relieve the developer from 95 percent of common data persistence related programming tasks.
-Hibernate may not be the best solution for data-centric applications that only use stored-procedures to implement the business logic in the database, it is most useful with object-oriented domain models and business logic in the Java-based middle-tier. However, Hibernate can certainly help you to **remove or encapsulate vendor-specific SQL code** and will help with the common task of result set translation from a tabular representation to a graph of objects.

Architecture





SessionFactory (org.hibernate.SessionFactory)
A threadsafe (immutable) cache of compiled mappings for a single database. A factory for Session and a client of ConnectionProvider. Might hold an optional (second-level) cache of data that is reusable between transactions, at a process- or cluster-level.

Session (org.hibernate.Session)
A single-threaded, short-lived object representing a conversation between the application and the persistent store. Wraps a JDBC connection. Factory for Transaction. Holds a mandatory (first-level) cache of persistent objects, used when navigating the object graph or looking up objects by identifier.

Persistent objects and collections
Short-lived, single threaded objects containing persistent state and business function. These might be ordinary JavaBeans/POJOs, the only special thing about them is that they are currently associated with (exactly one) Session. As soon as the Session is closed, they will be detached and free to use in any application layer (e.g. directly as data transfer objects to and from presentation).
Transient and detached objects and collections
Instances of persistent classes that are not currently associated with a Session. They may have been instantiated by the application and not (yet) persisted or they may have been instantiated by a closed Session.

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

ConnectionProvider (org.hibernate.connection.ConnectionProvider)
(Optional) A factory for (and pool of) JDBC connections. Abstracts application from underlying Datasource or DriverManager. Not exposed to application, but can be extended/implemented by the developer.

TransactionFactory (org.hibernate.TransactionFactory)
(Optional) A factory for Transaction instances. Not exposed to the application, but can be extended/ implemented by the developer.

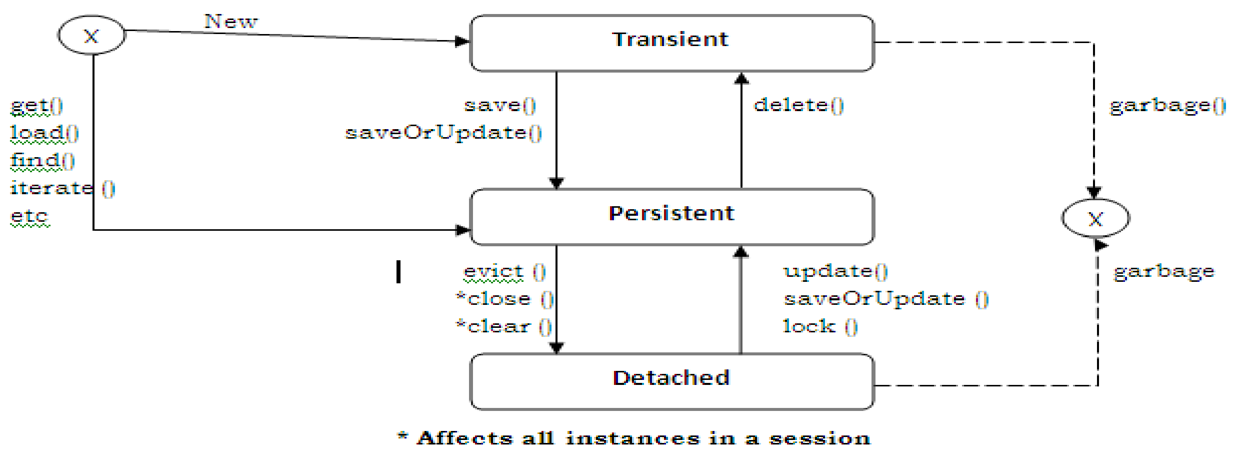
Extension Interfaces
Hibernate will usually let you plug in your own custom implementation by implementing an interface. Extension points include:
■ Primary key generation (IdentifierGenerator interface)
■ SQL dialect support (Dialect abstract class)
■ Caching strategies (Cache and CacheProvider interfaces)
■ JDBC connection management (ConnectionProvider interface)
■ Transaction management (TransactionFactory, Transaction, and TransactionManagerLookup interfaces)
■ ORM strategies (ClassPersister interface hierarchy)
■ Property access strategies (PropertyAccessor interface)
■ Proxy creation (ProxyFactory interface)

Note:
It's important to understand the difference in configuring Hibernate for managed and non-managed environments:
■ *Managed environment*—Pools resources such as database connections and allows transaction boundaries and security to be specified declaratively (that is, in metadata). A J2EE application server such as JBoss, BEA WebLogic, or IBM WebSphere implements the standard (J2EE-specific) managed environment for Java.
In managed environments, Hibernate integrates with container-managed transactions and datasources.

■ *Non-managed environment*—Provides basic concurrency management via thread pooling. A servlet container like Jetty or Tomcat provides a nonmanaged server environment for Java web applications. A stand-alone desktop or command-line application is also considered non-managed. Nonmanaged environments don't provide automatic transaction or resource management or security infrastructure. The application itself manages database connections and demarcates transaction boundaries.
In the case of a non-managed environment, Hibernate handles transactions and JDBC connections (or delegates to application code that handles these concerns).

Hibernate can be configured for deployment in both environments.
Note: *Method chaining*
- *Method chaining* is a programming style supported by many Hibernate interfaces because these interfaces return the receiving object. { more difficult to debug} .It's better to write each method invocation on a different line. Otherwise, it might be difficult to step through the code in your debugger.

Session object life cycle :



The Hibernate Session object is the persistence context:

Transient	**Persistent**	**Detached**
- The instance is not, and has never been associated with any persistence context. It has no persistent identity (primary key value).	- The instance is currently associated with a persistence context. It has a persistent identity. - For a particular persistence context, Hibernate *guarantees* that persistent identity is equivalent to Java identity.	- The instance was once associated with a persistence context, but that context was closed, or the instance was Serialized to another process. It has a persistent identity. - For detached instances, Hibernate makes *no guarantees* about the relationship between persistent identity and Java identity

Distinguishing between transient and detached instances

Hibernate will assume that an instance is an unsaved transient instance if:

■ The identifier property (if it exists) is `null`.

■ The version property (if it exists) is `null`.

■ You supply an `unsaved-value` in the mapping document for the class, and the value of the `identifier` property matches.

■ You supply an `unsaved-value` in the mapping document for the `version` property, and the value of the `version` property matches.

■ You supply a Hibernate `Interceptor` and return `Boolean.TRUE` from `Interceptor.isUnsaved()` after checking the instance in your code.

Session methods :

session.load()	session.get()	session.find()
session.save()	session.persist()	session.refresh()
session.update()	session.saveorUpdate()	session.merge()
session.evict()	session.remove()	
session.delete()	session.clear()	session.close()
session.contains()	session.Connection()	session.isConnected()
session.isOpened()	session.getSessinFactory()	

evict()	**delete()**
-will remove the object from session but object exists in the DB(Detached state)	-will remove the object from both session and DB

refresh()	**flush()**
-Synchronises database data with the session data. -executes SELECT query.	-Synchronises session data with the database data. -executes UPDATE query.
flush() -Synchronises session data with the database data. -executes only UPDATE query.	**commit()** -Synchronises session data with the database data. -executes UPDATE and COMMIT.

Note: Most of the cases we don't prefer to use flush() because it is done while calling commit().

For only one record :

evict()]	**replicate()**
P->D	**D->P**

save()	**persist()**
Return type: Serializable -Returns the Identifier value.	Return type: void -it won't return any value.

evict()	**clear()**
-This method removes one object from the session. (P->D).	-This method removes all the objects from the session. (P->D)

delete()	**clear()**
-will remove the object from both session and DB	- This method removes all the objects from the session.

evict()	**remove()**
-This method removes one object from the session. [only that particular record]	- This method removes all the objects from the session. [only that particular record]

isConnected()	**isOpened()**
-This method is used to check whether connected to database or not.	- This method is used to check whether connection is opened or not.

Usually update() or saveOrUpdate() are used in the following scenario:

• the application loads an object in the first session
• the object is passed up to the UI tier
• some modifications are made to the object
• the object is passed back down to the business logic tier
• the application persists these modifications by calling update() in a second session

saveOrUpdate() does the following:

• if the object is already persistent in this session, do nothing
• if another object associated with the session has the same identifier, throw an exception
• if the object has no identifier property, save() it
• if the object's identifier has the value assigned to a newly instantiated object, save() it
• 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
• otherwise update() the object

And
merge() is very different:
• 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
• if there is no persistent instance currently associated with the session, try to load it from the database, or create a new persistent instance
• the persistent instance is returned
• the given instance does not become associated with the session, it remains detached

replicate():
The ReplicationMode determines how replicate() will deal with conflicts with existing rows in the database.

• ReplicationMode.IGNORE - ignore the object when there is an existing database row with the same identifier
• ReplicationMode.OVERWRITE - overwrite any existing database row with the same identifier
• ReplicationMode.EXCEPTION - throw an exception if there is an existing database row with the same identifier
• ReplicationMode.LATEST_VERSION - overwrite the row if its version number is earlier than the version number of the object, or ignore the object otherwise

Usecases for this feature include reconciling data entered into different database instances, upgrading system configuration information during product upgrades, rolling back changes made during non-ACID transactions and more.

Configuration
The Configuration object is used to configure and bootstrap Hibernate. The application uses a Configuration instance to specify the location of mapping documents and Hibernate-specific properties and then create the SessionFactory.

SessionFactory :
-SessionFactory is a thread-safe global object, instantiated once. Once a `SessionFactory` is created, its mappings are immutable.

- Hibernate does allow your application to instantiate more than one SessionFactory. This is useful if you are using more than one database. Each SessionFactory is then available for one database and ready to produce Sessions to work with that particular database and a set of class mappings.

Note :
If you give the `SessionFactory` a name in your configuration file, Hibernate will in fact try to bind it to JNDI after it has been built. To avoid this code completely you could also use JMX deployment and let the JMX capable container instantiate and bind a `HibernateService` to JNDI.

- In a J2EE environment, a `SessionFactory` bound to JNDI is easily shared between different threads and between various Hibernate-aware components.

-Of course,JNDI isn't the only way that application components might obtain a `SessionFactory`.Including use of the `ServletContext` or a `static final` variable in a singleton.

-The `SessionFactory` will automatically bind itself to JNDI if the property `hibernate.session_factory_name` is set to the name of the directory node. If your runtime environment doesn't provide a default JNDI context you need to specify a JNDI initial context using the properties `hibernate.jndi.url` and `hibernate.jndi.class`.

Session:
-A session represents a single threaded unit of work.
-A session begins when it is first needed, when the first call to getCurrentSession () is made. It is then bounded by hibernate to the current thread. When the transaction ends either committed (or) rollback. Hibernate also unbinds the session from the thread and closes it for you.

Q.How many ways can we pass configuration properties to Hibernate ?.
The various options include:
1. Pass an instance of java.util.Properties to Configuration.setProperties().

2. Place hibernate.properties in a root directory of the classpath.

3. Set System properties using java -Dproperty=value.

4. Include <property> elements in hibernate.cfg.xml {**Most preferred**}

5.
```
Configuration cfg = new Configuration()
.addClass(org.hibernate.auction.Item.class)
.addClass(org.hibernate.auction.Bid.class)
.setProperty("hibernate.dialect", "org.hibernate.dialect.MySQLInnoDBDialect")
.setProperty("hibernate.connection.datasource", "java:comp/env/jdbc/test")
.setProperty("hibernate.order_updates", "true");
```

6.
```
Configuration cfg = new Configuration()
.addResource("Item.hbm.xml")
.addResource("Bid.hbm.xml");
```
Note:
- You load multiple mapping files by calling addResource()as often as you have to. Alternatively, you can use the method addClass(), passing a persistent class as the parameter.

```
SessionFactory sessions = new Configuration()
.addResource("hello/Message.hbm.xml")
.setProperties( System.getProperties() )
.buildSessionFactory();

(or)

SessionFactory sessions = new Configuration()
.addClass(org.hibernate.auction.model.Item.class)
.addClass(org.hibernate.auction.model.Category.class)
.addClass(org.hibernate.auction.model.Bid.class)
.setProperties( System.getProperties() )
.buildSessionFactory();
```

Hibernate Mapping file (hbm.xml):
-XML mapping files *must* be placed in the classpath.-The mapping file tells Hibernate what table is the database it has to access and what columns in that table it should use.

-- By convention,Hibernate XML mapping files are named with the .hbm.xml extension.
-Another convention is to have one mapping file per class

- By default, no properties of the class are considered persistent

-Hibernate uses reflection on the mapped class to help determine the defaults.

-Hibernate uses reflection to determine the Java type of the property.

Note:
-Hibernate will not load the DTD file from the web, but first it looks up from classpath of the Application.
-The DTD file is included in *hibernate3.jar* as well as src directory of the hibernate distribution.
-Hibernate automatically looks for a file called *hibernate.cfg.xml* in the root of the classpath, on startup.
- copy *log4j.properties* from the Hibernate distribution (it's in the *etc/* directory) to your *src* directory.
- All Hibernate property names and semantics are defined on the class org.hibernate.cfg.**Environment**.
- `hibernate.cfg.xml` file can be used as a replacement for the `hibernate.properties` file or, if both are present, to override properties.

Note:
<session-factory> -- a global factory responsible for a particular database.

-If you have several databases, use several <session-factory> configurations, usually in several configuration files

-The *hbm2ddl.auto* option turns on automatic generation of database schemas - directly into the database. The *hbm2ddl* creates the database schema on the first run, and subsequent application restarts will use this schema. If you change the mapping and/or database schema, you have to re-enable *hbm2ddl* once again. It is usually turned on in continuous unit testing.

Q.What are the Advantages of using hibernate.cfg.xml file over hibernate.properties ?
a. externalization of the mapping file names to configuration.
b. It is more convenient to tune the Hibernate cache.

```
SessionFactory sf = new Configuration().configure().buildSessionFactory();
                                    {takes hibernate.cfg.xml file }
(Or)

SessionFactory sf = new
Configuration().configure("catdb.cfg.xml").buildSessionFactory();
```

Note:
When `configure()` was called, Hibernate searched for a file named hibernate.cfg.xml in the classpath. If you wish to use a different filename or have Hibernate look in a subdirectory, you must pass a path to the `configure()` method:

```
SessionFactory sessions = new Configuration()
.configure("/hibernate-config/auction.cfg.xml")
.buildSessionFactory();
```

Q.How to obtain SessionFactory from Configuration object ?
```
SessionFactory sessions = cfg.buildSessionFactory();

Session session = sessions.openSession(); // open a new Session
```

Note: -Once the Transaction object is created it is available with session so whenever the Transaction is required ,we can get it using getTransaction().

Q.How to get the Connection object from hibernate session object?.
Connection connection=session.Connnection();

Q.How to get the SessionFactory from hibernate session object?.
session.getSessionFactory();

Q.How to get the Transaction from hibernate session object?.
session.getTransacition();
session.getTransaction.commit()
session.getTransaction.rollback()

Note:
-Hibernate is the layer in your application which connects to this database, so it needs connection information. The connections are made through a JDBC connection pool, which we also have to configure.

-Hibernate will obtain (and pool) connections using *java.sql.DriverManager* if you set the following properties:

Hibernate JDBC Properties

Property name	Purpose
`hibernate.connection.driver_class`	*jdbc driver class*
`hibernate.connection.url`	*jdbc URL*
`hibernate.connection.username`	*database user*
`hibernate.connection.password`	*database user password*
`hibernate.connection.pool_size`	*maximum number of pooled connections*

Note:
-Hibernate internal connection Pool is not intended in Production systems or even in Performance testing.We should use 3rd party Pool for best performance and stability.

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

Here is an example `hibernate.properties` file for C3P0:
```
hibernate.connection.driver_class = org.postgresql.Driver
hibernate.connection.url = jdbc:postgresql://localhost/mydatabase
hibernate.connection.username = myuser
hibernate.connection.password = secret
hibernate.c3p0.min_size=5
hibernate.c3p0.max_size=20
hibernate.c3p0.timeout=1800
hibernate.c3p0.max_statements=50
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

Note: Hibernate Datasource Properties

Propery name	Purpose
`hibernate.connection.datasource`	*datasource JNDI name*
`hibernate.jndi.url`	*URL of the JNDI provider* (optional)
`hibernate.jndi.class`	*class of the JNDI* `InitialContextFactory` (optional)
`hibernate.connection.username`	*database user* (optional)
`hibernate.connection.password`	*database user password* (optional)

Here's an example `hibernate.properties` file for an application server provided JNDI datasource:
```
hibernate.connection.datasource = java:/comp/env/jdbc/test
hibernate.transaction.factory_class = \org.hibernate.transaction.JTATransactionFactory
hibernate.transaction.manager_lookup_class =
\org.hibernate.transaction.JBossTransactionManagerLookup
hibernate.dialect = org.hibernate.dialect.PostgreSQLDialect
```

JDBC connections obtained from a JNDI datasource will automatically participate in the container-managed transactions of the application server.

<u>Note</u>:Hibernate Configuration Properties

hibernate.dialect	***eg.*** **full.classname.of.Dialect**
hibernate.show_sql	***eg.*** **true | false**
hibernate.format_sql	***eg.*** **true | false**
hibernate.default_schema	***eg.*** **SCHEMA_NAME**
hibernate.default_catalog	***eg.*** **CATALOG_NAME**
hibernate.generate_statistics	***eg.*** **true | false**
hibernate.session_factory_name	***eg.*** **jndi/composite/name**
hibernate.max_fetch_depth	*eg.* recommended values between 0 and 3
hibernate.default_batch_fetch_size	*eg.* recommended values 4, 8, 16
hibernate.default_entity_mode	dynamic-map, dom4j, pojo
hibernate.order_updates	*eg.* true | false
hibernate.use_identifer_rollback	*eg.* true | false
hibernate.use_sql_comments	*eg.* true | **false**

<u>Note:</u>
-Setting the property `hibernate.show_sql` to the value `true` enables logging of all generated SQL to the console. You'll use it for troubleshooting, performance tuning, and just to see what's going on. It pays to be aware of what your ORM layer is doing.

<u>Note:</u> Hibernate JDBC and Connection Properties

hibernate.connection.*<propertyName>*	
hibernate.jndi.*<propertyName>*	
hibernate.connection.autocommit	***eg.*** **true | false**
hibernate.connection.release_mode	***eg.*** **on_close|after_transaction|after_statement | auto**
hibernate.cache.use_query_cache	***eg.*** **true|false**
hibernate.cache.use_second_level_cache	***eg.*** **true|false**
hibernate.jdbc.fetch_size	
hibernate.jdbc.batch_size	*eg.* recommended values between 5 and 30
hibernate.jdbc.batch_versioned_data	*eg.* true | **false**
hibernate.jdbc.factory_class	
hibernate.jdbc.use_scrollable_resultset	*eg.* true | false
hibernate.jdbc.use_streams_for_binary	*eg.* true | false
hibernate.jdbc.use_get_generated_keys	*eg.* true|false
hibernate.connection.provider_class	*eg.* classname.of.ConnectionProvider
hibernate.connection.isolation	*eg.* 1, 2, 4, 8
hibernate.cache.provider_class	*eg.* classname.of.CacheProvider
hibernate.cache.use_minimal_puts	*eg.* true|false
hibernate.cache.query_cache_factory	*eg.* classname.of.QueryCache
hibernate.cache.region_prefix	*eg.* Prefix
hibernate.cache.use_structured_entries	*eg.* true|false

<u>Note:</u> The properties prefixed by hibernate.cache allow you to use a process or cluster scoped second-level cache system with Hibernate.

<u>Note:</u> Hibernate Transaction Properties

hibernate.transaction.flush_before_completion	***eg.*** **true | false**
hibernate.transaction.auto_close_session	***eg.*** **true | false**
hibernate.transaction.factory_class	***eg.*** **classname.of.TransactionFactory**
jta.UserTransaction	*eg.* jndi/composite/name
hibernate.transaction.manager_lookup_class	*eg.* classname.of.TransactionManagerLookup

Note: Miscellaneous Properties

hibernate.hbm2ddl.auto	***eg.* validate | update | create | create-drop**
hibernate.query.substitutions	***eg.* hqlLiteral=SQL_LITERAL, hqlFunction=SQLFUNC**
`hibernate.current_session_context_class`	*eg.* `jta | thread | custom.Class`
`hibernate.query.factory_class`	*eg.*`org.hibernate.hql.ast.ASTQueryTranslatorFactory` or `org.hibernate.hql.classic.ClassicQueryTranslatorFactory`
`hibernate.cglib.use_reflection_optimizer`	*eg.* `true | false`

Hibernate statistics:

If you enable hibernate.generate_statistics, Hibernate will expose a number of metrics that are useful when tuning a running system via SessionFactory.getStatistics().Hibernate can even be configured to expose these statistics via JMX.

Logging :

The commons-logging service will direct output to either Apache Log4j. To use Log4j you will need to place a log4j.properties file in your classpath, an example properties file is distributed with Hibernate in the *src/* directory.

-Hibernate executes SQL statements *asynchronously*. You can use logging to get a view of Hibernate's internals.

-`hibernate.show_sql` configuration parameter, which is usually the first port of call when troubleshooting.

-To see any output from log4j, you'll need a file named log4j.properties in your classpath.

-With this configuration, you won't see many log messages at runtime. Replacing *info* with *debug* for the `log4j.logger.net.sf.hibernate` category will reveal the inner workings of Hibernate. Make sure you don't do this in a production environment—writing the log will be much slower than the actual database access.

Hibernate Log Categories

org.hibernate.SQL	Log all SQL DML statements as they are executed.
org.hibernate.type	Log all JDBC parameters.
org.hibernate.tool.hbm2ddl	Log all SQL DDL statements as they are executed.
org.hibernate.pretty	Log the state of all entities (max 20 entities) associated with the session at flush time
org.hibernate.cache	Log all second-level cache activity
org.hibernate.transaction	Log transaction related activity
org.hibernate.jdbc	Log all JDBC resource acquisition
org.hibernate.hql.ast.AST	Log HQL and SQL ASTs during query parsing
org.hibernate.secure	Log all JAAS authorization requests
org.hibernate	Log everything (a lot of information, but very useful for troubleshooting)

When developing applications with Hibernate, you should almost always work with ***debug*** enabled for the category org.hibernate.SQL, or, alternatively, the property **hibernate.show_sql** enabled.

Note:

Both the `hibernate.properties` and the `hibernate.cfg.xml` files provide the same function: to configure Hibernate. Using an XML configuration file is certainly more comfortable than a properties file or even programmatic property configuration.

-If you have both `hibernate.properties` and `hibernate.cfg.xml` in the classpath,the settings of the XML configuration file will override the settings used in the properties.

Note:
-`hibernate.properties`, contains only configuration parameters and we need programmatically set the mapping documents.
-`hibernate.cfg.xml` file may also specify the location of mapping documents and configuration parameters,.

- You declare the properties in a file named `hibernate.properties`, so you need only place this file in the application classpath. It will be automatically detected and read when Hibernate is first initialized when you create a `Configuration` object.

-Finally, you have the `hibernate.properties`, `hibernate.cfg.xml`, and log4j.properties configuration files.

Implementing a NamingStrategy :
The interface ***org.hibernate.cfg.NamingStrategy*** allows you to specify a "naming standard" for database objects and schema elements.

The default strategy used by Hibernate is quite minimal.

You may specify a different strategy by calling Configuration.setNamingStrategy() before adding mappings. Example :

```
SessionFactory sf = new Configuration()
.setNamingStrategy(ImprovedNamingStrategy.INSTANCE)
.addFile("Item.hbm.xml")
.addFile("Bid.hbm.xml")
.buildSessionFactory();
```

org.hibernate.cfg.ImprovedNamingStrategy is a built-in strategy that might be a useful starting point for some applications.

--we can implement Hibernate's `NamingStrategy` interface

```
public class CENamingStrategy implements NamingStrategy
{
            public String classToTableName(String className)
            {
                  return tableName(StringHelper.unqualify(className).toUpperCase() );
            }
            public String propertyToColumnName(String propertyName)
            {
                  return propertyName.toUpperCase();
            }
            public String tableName(String tableName)
            {
                  return "CE_" + tableName;
            }
            public String columnName(String columnName)
            {
            return columnName;
            }
            public String propertyToTableName(String className,String propertyName)
            {
                  return classToTableName(className) + '_'
            +propertyToColumnName(propertyName);
            }
}
```

-To activate a specific naming strategy, we can pass an instance to the Hibernate Configuration at runtime:

```
Configuration cfg = new Configuration();
```

```
cfg.setNamingStrategy( new CENamingStrategy() );
SessionFactory sessionFactory =
cfg.configure().buildSessionFactory();
```

Note:*Using quoted SQL identifiers*
-By default, Hibernate doesn't quote table and column names in the generated SQL.

-You may force Hibernate to quote an identifier in the generated SQL by enclosing the table or column name in backticks in the mapping document.

```
<class name="LineItem" table="`Line Item`">
    <id name="id" column="`Item Id`"/><generator class="assigned"/></id>
    <property name="itemNumber" column="`Item #`"/>
    ...
</class>
```

-If you quote a table or column name with backticks in the mapping document, Hibernate will always quote this identifier in the generated SQL.

Note:
-The value of a derived property is calculated at runtime by evaluation of an expression. You define the expression using the *formula* attribute.

-Formulas may refer to columns of the database table, call SQL functions, and include SQL subselects.

```
<property name="averageBidAmount"
        formula="( select AVG(b.AMOUNT) from BID b where b.ITEM_ID = ITEM_ID )"
        type="big_decimal"/>
```

Notice that unqualified column names refer to table columns of the class to which the derived property belongs.

Note:
-The following property never has its state written to the database:

```
            <property name="name"
            column="NAME"
            type="string"
            insert="false"
            update="false"/>
```

The property name of the JavaBean is therefore *immutable* and can be read from the database but not modified in any way. If the complete class is immutable, set the `immutable="false"` in the class mapping.

-In addition, the `dynamic-insert` attribute tells Hibernate whether to include unmodified property values in an SQL INSERT, and the `dynamic-update` attribute tells Hibernate whether to include unmodified properties in the SQL UPDATE:

```
            <class name="org.hibernate.auction.model.User"
            dynamic-insert="true"
            dynamic-update="true">
            ...
            </class>
```

These are both class-level settings. Enabling either of these settings will cause Hibernate to generate some SQL at runtime, instead of using the SQL cached at startup time.

Note : Mapping a class more than once
It is possible to provide more than one mapping for a particular persistent class. Hibernate lets you specify the entity name when working with persistent objects, when writing queries, or when mapping associations to the named entity.

J2EE Application Server integration :

Hibernate has the following integration points for J2EE infrastructure:
• *Container-managed datasources*: Hibernate can use JDBC connections managed by the container and provided through JNDI. Usually, a JTA compatible TransactionManager and a ResourceManager take care of transaction management (CMT), esp. distributed transaction handling across several datasources. You may of course also demarcate transaction boundaries programatically (BMT) or you might want to use the optional Hibernate Transaction API for this to keep your code portable.

• *JTA Session binding:* The Hibernate Session may be automatically bound to the scope of JTA transactions. Simply lookup the SessionFactory from JNDI and get the current Session. Let Hibernate take care of flushing and closing the Session when your JTA transaction completes. Transaction demarcation is either declarative (CMT) or programmatic (BMT/UserTransaction).

• *JMX deployment:* If you have a JMX capable application server (e.g. JBoss AS), you can chose to deploy Hibernate as a managed MBean. This saves you the one line startup code to build your SessionFactory from a Configuration. The container will startup your HibernateService, and ideally also take care of service dependencies (Datasource has to be available before Hibernate starts, etc).

Transaction strategy configuration :
The Hibernate Session API is independent of any transaction demarcation system in your architecture.

You have to specify a factory class for Transaction instances by setting the Hibernate configuration property

hibernate.transaction.factory_class.

There are three standard (built-in) choices:
1.`org.hibernate.transaction.JDBCTransactionFactory`
delegates to database (JDBC) transactions (default)

2.`org.hibernate.transaction.JTATransactionFactory`
delegates to container-managed transaction if an existing transaction is underway in this context (e.g. EJB session bean method), otherwise a new transaction is started and bean-managed transaction are used.

3.`org.hibernate.transaction.CMTTransactionFactory`
delegates to container-managed JTA transactions

4.You may also define your own transaction strategies (for a CORBA transaction service, for example).

Note:
Some features in Hibernate (i.e. the second level cache, Contextual Sessions with JTA, etc.) require access to the JTA TransactionManager in a managed environment.

JTA TransactionManagers

Transaction Factory	**Application Server**
`org.hibernate.transaction.JBossTransactionManagerLookup`	JBoss
`org.hibernate.transaction.WeblogicTransactionManagerLookup`	Weblogic
`org.hibernate.transaction.WebSphereTransactionManagerLookup`	WebSphere
`org.hibernate.transaction.WebSphereExtendedJTATransactionLookup`	WebSphere 6

JNDI-bound SessionFactory :

A JNDI bound Hibernate SessionFactory can simplify the lookup of the factory and the creation of new Sessions. Note that this is not related to a JNDI bound Datasource, both simply use the same registry!

If you wish to have the SessionFactory bound to a JNDI namespace, specify a name (eg. *java:hibernate/SessionFactory*) using the property hibernate.session_factory_name. If this property is omitted, the SessionFactory will not be bound to JNDI. (This is especially useful in environments with a readonly JNDI default implementation, e.g. Tomcat.)

When binding the SessionFactory to JNDI, Hibernate will use the values of *hibernate.jndi.url, hibernate.jndi.class* to instantiate an initial context. If they are not specified, the default InitialContext will be used.

Hibernate will automatically place the SessionFactory in JNDI after you call *cfg.buildSessionFactory().*This means you will at least have this call in some startup code (or utility class) in your application, unless you use JMX deployment with the HibernateService .

If you use a JNDI SessionFactory, an EJB or any other class may obtain the SessionFactory using a JNDI lookup.
It is recommend that you bind the SessionFactory to JNDI in a managend environment and use a static singleton otherwise.

Current Session context management with JTA :

Using the "jta" session context, if there is no Hibernate Session associated with the current JTA transaction, one will be started and associated with that JTA transaction the first time you call `sessionFactory.getCurrentSession()`. The Sessions retrieved via `getCurrentSession()` in "jta" context will be set to automatically flush before the transaction completes, close after the transaction completes, and aggressively release JDBC connections after each statement. This allows the Sessions to be managed by the lifecycle of the JTA transaction to which it is associated, keeping user code clean of such management concerns. Your code can either use JTA programmatically through UserTransaction, or (recommended for portable code) use the Hibernate Transaction API to set transaction boundaries. If you run in an EJB container, declarative transaction demarcation with CMT is preferred.

*Java Management Extensions (JMX)*deployment :

JBoss is a good open source starting point. All services (even the EJB container) in JBoss are implemented as MBeans and can be managed via a supplied console interface.

The JMX specification defines the following components:
- *The JMX MBean*—A reusable component (usually infrastructural) that exposes an interface for *management* (administration)
- *The JMX container*—Mediates generic access (local or remote) to the MBean
- *The (usually generic) JMX client*—May be used to administer any MBean via the JMX container

Note :

-An application server with support for JMX (such as JBoss) acts as a JMX container and allows an MBean to be configured and initialized as part of the application server startup process. It's possible to monitor and administer the MBean using the application server's administration console.

-An MBean may be packaged as a JMX service, which is not only portable between application servers with JMX support but also deployable to a running system (a *hot deploy*).

-Hibernate may be packaged and administered as a JMX MBean. The Hibernate JMX service allows Hibernate to be initialized at application server startup and controlled (configured) via a JMX client.

-The biggest advantage of Hibernate with JMX is the automatic startup; it means you no longer have to create a `Configuration` and build a `SessionFactory` in your application code, but can simply access the `SessionFactory` through JNDI once the `HibernateService` has been deployed and started.

The line `cfg.buildSessionFactory()` still has to be executed somewhere to get a `SessionFactory` into JNDI.

You can do this either in a `static` initializer block (like the one in `HibernateUtil`) or you deploy Hibernate as a *managed service*.

Hibernate is distributed with org.hibernate.jmx.HibernateService for deployment on an application server with JMX capabilities, such as JBoss AS.

Persistent Classes:

- class uses standard JavaBean naming conventions for property getter and setter methods, as well as private visibility for the fields.

- The no-argument constructor is required to instantiate an object of this class through reflection.

- The no-argument constructor is a requirement for all persistent classes; Hibernate has to create objects for you, using Java Reflection. The constructor can be private, however, package visibility is required for runtime proxy generation and efficient data retrieval without bytecode instrumentation.

- The id property holds a unique identifier value for a particular event.

There are four main rules to follow here:

a. Implement a no-argument constructor
b. Provide an identifier property (optional)
c. Prefer non-final classes (optional)
d. Declare accessors and mutators for persistent fields (optional)

a.Implement a no-argument constructor :

All persistent classes must have a default constructor (which may be nonpublic) so that Hibernate can instantiate them using Constructor.newInstance(). We strongly recommend having a default constructor with at least *package* visibility for runtime proxy generation in Hibernate.

b.Provide an identifier property (optional) :

The identifier property is strictly optional.

In fact, some functionality is available only to classes which declare an identifier property:

• Transitive reattachment for detached objects (cascade update or cascade merge)
• Session.saveOrUpdate()
• Session.merge()

It is recommend you declare consistently-named identifier properties on persistent classes. We further recommend that you use a nullable (ie. non-primitive) type.

c. Prefer non-final classes (optional) :

-A central feature of Hibernate, *proxies*, depends upon the persistent class being either non-final, or the implementation of an interface that declares all public methods.

-You can persist `final` classes that do not implement an interface with Hibernate, but you won't be able to use proxies for lazy association fetching - which will limit your options for performance tuning.

-You should also avoid declaring `public final` methods on the non-final classes. If you want to use a class with a `public final` method, you must explicitly disable proying by setting `lazy="false"`.

d.Declare accessors and mutators for persistent fields (optional) :

-By default, Hibernate persists JavaBeans style properties, and recognizes method names of the form `getFoo`, `isFoo` and `setFoo`.

-Properties need *not* be declared public - Hibernate can persist a property with a *default*, *protected* or *private* get / set pair.

Note: 1.Implementing inheritance

- A subclass must also observe the first and second rules. It inherits its identifier property from the superclass.

2. Implementing equals() and hashCode()

You have to override the equals() and hashCode() methods if you

• intend to put instances of persistent classes in a Set (the recommended way to represent many-valued associations) *and*

• intend to use reattachment of detached instances

Note :

We recommend implementing `equals()` and `hashCode()` using *Business key equality*. Business key equality means that the `equals()` method compares only the properties that form the business key, a key that would identify our instance in the real world (a *natural* candidate key). A business key does not have to be as solid as a database primary key candidate. Immutable or unique properties are usually good candidates for a business key.

3.Dynamic models

-Persistent entities don't necessarily have to be represented as POJO classes or as JavaBean objects at runtime. Hibernate also supports dynamic models (using Maps of Maps at runtime) and the representation of entities as DOM4J trees.

-By default, Hibernate works in normal POJO mode. You may set a default entity representation mode for a particular `SessionFactory` using the `default_entity_mode` configuration option.

Advantage:

The advantages of a dynamic mapping are quick turnaround time for prototyping without the need for entity class implementation. However, you lose compile-time type checking and will very likely deal with many exceptions at runtime.

Note:

1. XDoclet, Middlegen and AndroMDA are the tools to generate generate the mapping document.

2. The actual DTD may be found at the URL above, in the directory `hibernate-x.x.x/src/org/hibernate` or in `hibernate3.jar`. Hibernate will always look for the DTD in its classpath first.

Note : hibernate-mapping

```
<hibernate-mapping
schema="schemaName"                              (1)
catalog="catalogName"                            (2)
default-cascade="cascade_style"                  (3)
default-access="field|property|ClassName"        (4)
default-lazy="true|false"                        (5)
auto-import="true|false"                         (6)
package="package.name"                           (7)
/>
```

(1) `schema` (optional): The name of a database schema.
(2) `catalog` (optional): The name of a database catalog.
(3) `default-cascade` (optional - defaults to `none`): A default cascade style.

(4) `default-access` (optional - defaults to `property`): The strategy Hibernate should use for accessing all properties. Can be a custom implementation of `PropertyAccessor`.

(5) `default-lazy` (optional - defaults to `true`): The default value for unspecifed *lazy* attributes of class and collection mappings.

(6) `auto-import` (optional - defaults to `true`): Specifies whether we can use unqualified class names (of classes in this mapping) in the query language.

(7) `package`(optional): Specifies a package prefix to assume for unqualified class names in the mapping document.

<u>Note :</u> a. If you have <u>two persistent classes with the same (unqualified) name</u>, you should set `auto-import="false"`. Hibernate will throw an exception if you attempt to assign two classes to the same "imported" name.

b.It is good practice to map only a single persistent class (or a single class hierarchy) in one mapping file and name it after the persistent superclass, e.g. `Cat.hbm.xml`,`Dog.hbm.xml`, or if using inheritance, `Animal.hbm.xml`.

<u>Note:</u> class

```
<class
name="ClassName"                                     (1)
table="tableName"                                    (2)
discriminator-value="discriminator_value"            (3)
mutable="true|false"                                 (4)
schema="owner"                                       (5)
catalog="catalog"                                    (6)
proxy="ProxyInterface"                               (7)
dynamic-update="true|false"                          (8)
dynamic-insert="true|false"                          (9)
select-before-update="true|false"                    (10)
polymorphism="implicit|explicit"                     (11)
where="arbitrary sql where condition"                (12)
persister="PersisterClass"                           (13)
batch-size="N"                                       (14)
optimistic-lock="none|version|dirty|all"             (15)
lazy="true|false"                                    (16)
entity-name="EntityName"                             (17)
check="arbitrary sql check condition"                (18)
rowid="rowid"                                        (19)
subselect="SQL expression"                           (20)
abstract="true|false"                                (21)
node="element-name"
/>
```

(1) `name` (optional): The fully qualified Java class name of the persistent class (or interface). If this attribute is
missing, it is assumed that the mapping is for a non-POJO entity.

(2) `table` (optional - defaults to the unqualified class name): The name of its database table.

(3) `discriminator-value` (optional - defaults to the class name): A value that distiguishes individual subclasses, used for polymorphic behaviour. Acceptable values include `null` and `not null`.

(4) `mutable` (optional, defaults to `true`): Specifies that instances of the class are (not) mutable.

(5) `schema` (optional): Override the schema name specified by the root `<hibernate-mapping>` element.

(6) `catalog` (optional): Override the catalog name specified by the root `<hibernate-mapping>` element.

(7) `proxy` (optional): Specifies an interface to use for lazy initializing proxies. You may specify the name of the class itself.

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

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

(10) `select-before-update` (optional, defaults to `false`): Specifies that Hibernate should *never* perform an SQL `UPDATE` unless it is certain that an object is actually modified. In certain cases (actually, only when a transient object has been associated with a new session using `update()`), this means that Hibernate will perform an extra SQL `SELECT` to determine if an `UPDATE` is actually required.

(11) `polymorphism` (optional, defaults to `implicit`): Determines whether implicit or explicit query polymorphism is used.

(12) `where` (optional) specify an arbitrary SQL `WHERE` condition to be used when retrieving objects of this class

(13) `persister` (optional): Specifies a custom `ClassPersister`.

(14) `batch-size` (optional, defaults to `1`) specify a "batch size" for fetching instances of this class by identifier.

(15) `optimistic-lock` (optional, defaults to `version`): Determines the optimistic locking strategy.

(16) `lazy` (optional): Lazy fetching may be completely disabled by setting `lazy="false"`.

(17) `entity-name` (optional, defaults to the class name): Hibernate3 allows a class to be mapped multiple times (to different tables, potentially), and allows entity mappings that are represented by Maps or XML at the Java level. In these cases, you should provide an explicit arbitrary name for the entity.

(18) `check` (optional): A SQL expression used to generate a multi-row *check* constraint for automatic schema generation.

(19) `rowid` (optional): Hibernate can use so called ROWIDs on databases which support. E.g. on Oracle, Hibernate can use the `rowid` extra column for fast updates if you set this option to `rowid`. A ROWID is an implementation detail and represents the physical location of a stored tuple.

(20) `subselect` (optional): Maps an immutable and read-only entity to a database subselect. Useful if you want to have a view instead of a base table, but don't.

(21) `abstract` (optional): Used to mark abstract superclasses in `<union-subclass>` hierarchies.

Note:
-Explicit polymorphism is useful when two different classes are mapped to the same table (this allows a "lightweight" class that contains a subset of the table columns).

-The `dynamic-update` and `dynamic-insert` settings are not inherited by subclasses and so may also be specified on the `<subclass>` or `<joined-subclass>` elements. These settings may increase performance in some cases, but might actually decrease performance in others.

-If you enable `dynamic-update`, you will have a choice of optimistic locking strategies:

- `version` check the version/timestamp columns
- `all` check all columns
- `dirty` check the changed columns, allowing some concurrent updates
- `none` do not use optimistic locking

-It *very* strongly recommend that you use version/timestamp columns for optimistic locking with Hibernate. This is the optimal strategy with respect to performance and is the only strategy that correctly handles modifications made to detached instances (ie. when `Session.merge()` is used).

<u>Note :</u>
<u>Scenario :1</u>
There is no difference between a view and a base table for a Hibernate mapping, as expected this is transparent at the database level (note that some DBMS don't support views properly, especially with updates). Sometimes you want to use a view, but can't create one in the database (ie. with a legacy schema). In this case, you can map an immutable and read-only entity to a given SQL subselect expression:

```
<class name="Summary">
<subselect>
select item.name, max(bid.amount), count(*)
from item
join bid on bid.item_id = item.id
group by item.name
</subselect>
<synchronize table="item"/>
<synchronize table="bid"/>
<id name="name"/>
...
</class>
```

Declare the tables to synchronize this entity with, <u>ensuring that auto-flush happens correctly</u>, and that queries against the derived entity <u>do not return stale data</u>. The `<subselect>` is available as both as an attribute and a nested mapping element.

<u>Note :</u> id
Mapped classes *must* declare the primary key column of the database table.

```
<id
name="propertyName"                                         (1)
type="typename"                                             (2)
column="column_name"                                        (3)
unsaved-value="null|any|none|undefined|id_value"            (4)
access="field|property|ClassName">                          (5)
node="element-name|@attribute-name|element/@attribute|."
<generator class="generatorClass"/>
</id>
```

(1) `name` (optional): The name of the identifier property.

(2) `type` (optional): A name that indicates the Hibernate type.

(3) `column` (optional - defaults to the property name): The name of the primary key column.

(4) `unsaved-value` (optional - defaults to a "sensible" value): An identifier property value that indicates that an instance is newly instantiated (unsaved), distinguishing it from detached instances that were saved or loaded in a previous session.

(5) `access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.

<u>Note :</u>
If the name attribute is missing, it is assumed that the class has no identifier property.

<u>Hibernate Mapping "type":</u>
-These are the Convertors which can translate from JAVA to SQL data types and vice versa.
- Hibernate will try to determine the correct conversion and mapping type itself if the **type** attribute is not present in the mapping.

<u>Note :</u> Generator
-names a Java class used to generate unique identifiers for instances of the persistent class.

- All generators implement the interface `org.hibernate.id.IdentifierGenerator` and override `generate()` method.

`increment`
generates identifiers of type `long`, `short` or `int` that are unique only when no other process is inserting data into the same table. *Do not use in a cluster.*

`identity`
supports identity columns in DB2, MySQL, MS SQL Server, Sybase and HypersonicSQL. The returned identifier is of type `long`, `short` or `int`.

`sequence`
uses a sequence in DB2, PostgreSQL, Oracle, SAP DB, McKoi or a generator in Interbase. The returned identifier is of type `long`, `short` or `int`

`hilo`
uses a hi/lo algorithm to efficiently generate identifiers of type `long`, `short` or `int`, given a table and column (by default `hibernate_unique_key` and `next_hi` respectively) as a source of hi values. The hi/lo algorithm generates identifiers that are unique only for a particular database.

`seqhilo`
uses a hi/lo algorithm to efficiently generate identifiers of type long, short or int, given a named database sequence.

`uuid`
uses a 128-bit UUID algorithm to generate identifiers of type string, unique within a network (the IP address is used). The UUID is encoded as a string of hexadecimal digits of length 32.

`guid`
uses a database-generated GUID string on MS SQL Server and MySQL.

`native`
picks `identity`, `sequence` or `hilo` depending upon the capabilities of the underlying database.

`assigned`
lets the application to assign an identifier to the object before `save()` is called. This is the default strategy if no `<generator>` element is specified.

`select`
retrieves a primary key assigned by a database trigger by selecting the row by some unique key and retrieving the primary key value.

`foreign`
uses the identifier of another associated object. Usually used in conjunction with a `<one-to-one>` primary key association.

Note :

-you can't use `hilo` when supplying your own `Connection` to Hibernate. When Hibernate is using an application server datasource to obtain connections enlisted with JTA, you must properly configure the `hibernate.transaction.manager_lookup_class`.

- The UUID contains: IP address, startup time of the JVM (accurate to a quarter second), system time and a counter value (unique within the JVM).

- `identity` and `sequence` style key generation,Both these strategies require two SQL queries to insert a new object.

- For cross-platform development, the `native` strategy will choose from the `identity`, `sequence` and `hilo` strategies, dependant upon the capabilities of the underlying database.

Note : composite-id

```
<composite-id
name="propertyName"
class="ClassName"
mapped="true|false"
access="field|property|ClassName">
node="element-name|."
<key-property name="propertyName" type="typename"
column="column_name"/>
<key-many-to-one name="propertyName class="ClassName"
column="column_name"/>
......
</composite-id>
```

-For a table with a composite key, you may map multiple properties of the class as identifier properties. The `<composite-id>` element accepts `<key-property>` property mappings and `<key-many-to-one>` mappings as child element

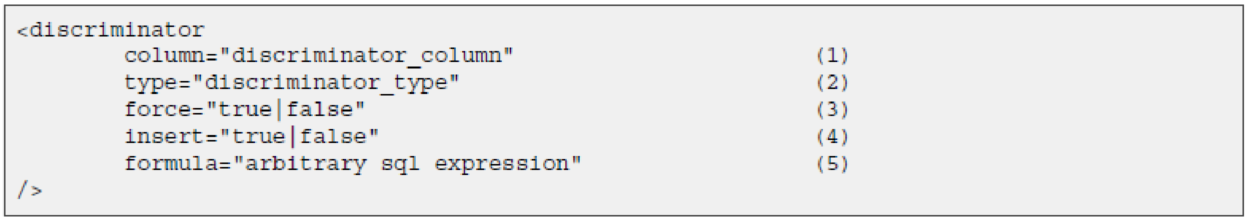
-Your persistent class *must* override `equals()` and `hashCode()` to implement composite identifier equality. It must also implements `Serializable`.

-*embedded* composite identifier
- a *mapped* composite identifier, where the identifier properties named inside the `<composite-id>` element are duplicated on both the persistent class and a separate identifier class.
[The disadvantage of this approach is quite obvious—code duplication.]

```
<composite-id class="MedicareId" mapped="true">
	<key-property name="medicareNumber"/>
	<key-property name="dependent"/>
</composite-id>
```

-an *identifier component* is the one best recommend for almost all applications.

Note : discriminator
The `<discriminator>` element is required for polymorphic persistence using the table-per-class-hierarchy mapping strategy and declares a discriminator column of the table. The discriminator column contains marker values that tell the persistence layer what subclass to instantiate for a particular row.



```
<discriminator
        column="discriminator_column"                      (1)
        type="discriminator_type"                          (2)
        force="true|false"                                 (3)
        insert="true|false"                                (4)
        formula="arbitrary sql expression"                 (5)
/>
```

(1) `column` (optional - defaults to `class`) the name of the discriminator column.
(2) `type` (optional - defaults to string) a name that indicates the Hibernate type
(3) `force` (optional - defaults to `false`) "force" Hibernate to specify allowed discriminator values even when retrieving all instances of the root class.
(4) `insert` (optional - defaults to true) set this to `false` if your discriminator column is also part of a mapped composite identifier. (Tells Hibernate to not include the column in SQL `INSERT`s.)
(5) `formula` (optional) an arbitrary SQL expression that is executed when a type has to be evaluated. Allows content-based discrimination.
Using the `formula` attribute you can declare an arbitrary SQL expression that will be used to evaluate the type of a row:

```
<discriminator
formula="case when CLASS_TYPE in ('a', 'b', 'c') then 0 else 1 end"
type="integer"/>
```

<u>Note :</u> version (optional)
-The `<version>` element is optional and indicates that the table contains versioned data. This is particularly useful if you plan to use ***long transactions.***

- Version numbers may be of Hibernate type `long`, `integer`, `short`, `timestamp` or `calendar`.

- A version or timestamp property <u>should never be null for a **detached** instance</u>, so <u>Hibernate will detact any instance with a **null version** or **timestamp** as **transient**,</u> no matter what other `unsaved-value` strategies are specified. ***Declaring a nullable version or timestamp property is an easy way to avoid any problems with transitive reattachment in Hibernate, especially useful for people using assigned identifiers or composite keys!***

- `<timestamp>` is equivalent to `<version type="timestamp">`. And
`<timestamp use-db="true">` is equivalent to `<version type="dbtimestamp">`

<u>Note :</u> property

```
<property
name="propertyName"                                         (1)
column="column_name"                                        (2)
type="typename"                                             (3)
update="true|false"                                         (4)
insert="true|false"                                         (4)
formula="arbitrary SQL expression"                          (5)
access="field|property|ClassName"                           (6)
lazy="true|false"                                           (7)
unique="true|false"                                         (8)
not-null="true|false"                                       (9)
optimistic-lock="true|false"                                (10)
generated="never|insert|always"                             (11)
node="element-name|@attribute-name|element/@attribute|."
index="index_name"
unique_key="unique_key_id"
length="L"
precision="P"
scale="S"
/>
```

(1) `name`: the name of the property, with an initial lowercase letter.

(2) `column` (optional - defaults to the property name): the name of the mapped database table column. This may also be specified by nested `<column>` element(s).

(3) `type` (optional): a name that indicates the Hibernate type.

(4) `update, insert` (optional - defaults to `true`) : specifies that the mapped columns should be included in SQL `UPDATE` and/or `INSERT` statements. Setting both to `false` allows a pure "derived" property whose value is initialized from some other property that maps to the same colum(s) or by a trigger or other application.

(5) `formula` (optional): an SQL expression that defines the value for a *computed* property. Computed properties
do not have a column mapping of their own.

(6) `access` (optional - defaults to `property`): The strategy Hibernate should use for accessing the property value.

(7) `lazy` (optional - defaults to `false`): Specifies that this property should be fetched lazily when the instance variable is first accessed (requires build-time bytecode instrumentation).

(8) `unique` (optional): Enable the DDL generation of a unique constraint for the columns. Also, allow this to be the target of a `property-ref`.

(9) `not-null` (optional): Enable the DDL generation of a nullability constraint for the columns.

(10) `optimistic-lock` (optional - defaults to `true`): Specifies that updates to this property do or do not require acquisition of the optimistic lock. In other words, determines if a version increment should occur when this property is dirty.

(11) `generated` (optional - defaults to `never`): Specifies that this property value is actually generated by the database.

typename could be:
1. The name of a Hibernate basic type (eg. `integer, string, character, date, timestamp, float, binary, serializable, object, blob`).
2. The name of a Java class with a default basic type (eg. `int, float, char, java.lang.String, java.util.Date, java.lang.Integer, java.sql.Clob`).
3. The name of a serializable Java class.
4. The class name of a custom type (eg. `com.illflow.type.MyCustomType`).

Note:
If you do not specify a type, Hibernate will use reflection upon the named property to take a guess at the correct Hibernate type. Hibernate will try to interpret the name of the return class of the property getter using rules 2, 3,4 in that order.

Note:
-You may specify your own strategy for property access by naming a class that implements the interface `org.hibernate.property.PropertyAccessor`.

-An especially powerful feature are derived properties. These properties are by definition read-only, the property value is computed at load time. You declare the computation as a SQL expression, this translates to a `SELECT` clause subquery in the SQL query that loads an instance

```
<property name="totalPrice"
      formula="( SELECT SUM (li.quantity*p.price) FROM LineItem li, Product p
                  WHERE li.productId = p.productId
                  AND li.customerId = customerId
                  AND li.orderNumber = orderNumber )"/>
```

- Note that you can reference the entities own table by not declaring an alias on a particular column.

Note: many-to-one

```
<many-to-one
        name="propertyName"                                          (1)
        column="column_name"                                         (2)
        class="ClassName"                                            (3)
        cascade="cascade_style"                                      (4)
        fetch="join|select"                                          (5)
        update="true|false"                                          (6)
        insert="true|false"                                          (6)
        property-ref="propertyNameFromAssociatedClass"               (7)
        access="field|property|ClassName"                            (8)
        unique="true|false"                                          (9)
        not-null="true|false"                                        (10)
        optimistic-lock="true|false"                                 (11)
        lazy="proxy|no-proxy|false"                                  (12)
        not-found="ignore|exception"                                 (13)
        entity-name="EntityName"                                     (14)
        formula="arbitrary SQL expression"                           (15)
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        index="index_name"
        unique_key="unique_key_id"
        foreign-key="foreign_key_name"
/>
```

Setting a value of the `cascade` attribute to any meaningful value other than `none` will propagate certain operations to the associated object. The meaningful values are the names of Hibernate's basic operations, `persist,merge, delete, save-update, evict, replicate, lock, refresh`, as well as the special values `deleteorphan` and `all` and comma-separated combinations of operation names, for example, `cascade="persist,merge,evict"` or `cascade="all,delete-orphan"`.

Note single valued associations (many-to-one and one-to-one associations) do not support orphan delete.

Note: one-to-one

```
<one-to-one
        name="propertyName"                                          (1)
        class="ClassName"                                            (2)
        cascade="cascade_style"                                      (3)
        constrained="true|false"                                     (4)
        fetch="join|select"                                          (5)
        property-ref="propertyNameFromAssociatedClass"               (6)
        access="field|property|ClassName"                            (7)
        formula="any SQL expression"                                 (8)
        lazy="proxy|no-proxy|false"                                  (9)
        entity-name="EntityName"                                     (10)
        node="element-name|@attribute-name|element/@attribute|."
        embed-xml="true|false"
        foreign-key="foreign_key_name"
/>
```

`constrained` (optional) specifies that a foreign key constraint on the primary key of the mapped table references the table of the associated class. This option affects the order in which `save()` and `delete()` are cascaded, and determines whether the association may be proxied (it is also used by the schema export tool).

`lazy` (optional - defaults to `proxy`): By default, single point associations are proxied.
`lazy="no-proxy"` specifies that the property should be fetched lazily when the instance variable is first accessed (requires build-time bytecode instrumentation).
`lazy="false"` specifies that the association will always be eagerly fetched.

Note that if `constrained="false"`, proxying is impossible and Hibernate will eager fetch the association!

There are two varieties of one-to-one association:
• primary key associations
• unique foreign key associations

Note: component, dynamic-component

The `<component>` element maps properties of a child object to columns of the table of a parent class. Components may, in turn, declare their own properties, components or collections.

```
<component
        name="propertyName"                 (1)
        class="className"                   (2)
        insert="true|false"                 (3)
        update="true|false"                 (4)
        access="field|property|ClassName"   (5)
        lazy="true|false"                   (6)
        optimistic-lock="true|false"        (7)
        unique="true|false"                 (8)
        node="element-name|."
>

        <property ...../>
        <many-to-one .... />
        ........
</component>
```

The `<dynamic-component>` element allows a `Map` to be mapped as a component, where the property names refer to keys of the map.

Note: properties

The `<properties>` element allows the definition of a named, logical grouping of properties of a class. The most important use of the construct is that it allows a combination of properties to be the target of a `property-ref`. It is also a convenient way to define a multi-column unique constraint.

```
<properties
        name="logicalName"                  (1)
        insert="true|false"                 (2)
        update="true|false"                 (3)
        optimistic-lock="true|false"        (4)
        unique="true|false"                 (5)
>

        <property ...../>
        <many-to-one .... />
        ........
</properties>
```

For example, if we have the following `<properties>` mapping:

```
<class name="Person">
    <id name="personNumber"/>
    ...
    <properties name="name"
            unique="true" update="false">
        <property name="firstName"/>
        <property name="initial"/>
        <property name="lastName"/>
    </properties>
</class>
```

Note : subclass

```
<subclass
        name="ClassName"                                  (1)
        discriminator-value="discriminator_value"         (2)
        proxy="ProxyInterface"                            (3)
        lazy="true|false"                                 (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        entity-name="EntityName"
        node="element-name"
        extends="SuperclassName">

        <property .... />
        .....
</subclass>
```

Note : joined-subclass

```
<joined-subclass
        name="ClassName"                     (1)
        table="tablename"                    (2)
        proxy="ProxyInterface"               (3)
        lazy="true|false"                    (4)
        dynamic-update="true|false"
        dynamic-insert="true|false"
        schema="schema"
        catalog="catalog"
        extends="SuperclassName"
        persister="ClassName"
        subselect="SQL expression"
        entity-name="EntityName"
        node="element-name">

        <key .... >

        <property .... />
        .....
</joined-subclass>
```

Note : key

```
<key
        column="columnname"                      (1)
        on-delete="noaction|cascade"             (2)
        property-ref="propertyName"              (3)
        not-null="true|false"                    (4)
        update="true|false"                      (5)
        unique="true|false"                      (6)
/>
```

We recommend that for systems where delete performance is important, all keys should be defined `on-delete="cascade"`, and Hibernate will use a database-level `ON CASCADE DELETE` constraint, instead of many individual `DELETE` statements. Be aware that this feature bypasses Hibernate's usual optimistic locking strategy for versioned data.

The `not-null` and `update` attributes are useful when mapping a unidirectional one to many association. If you map a unidirectional one to many to a non-nullable foreign key, you *must* declare the key column using `<key not-null="true">`.

Note : column and formula elements

```
<column name="column_name"
       length="N"
       precision="N"
       scale="N"
       not-null="true|false"
       unique="true|false"
       unique-key="multicolumn_unique_key_name"
       index="index_name"
       sql-type="sql_type_name"
       check="SQL expression"
       default="SQL expression"/>
```

Note : import

Suppose your application has two persistent classes with the same name, and you don't want to specify the fully qualified (package) name in Hibernate queries. Classes may be "imported" explicitly, rather than relying upon `auto-import="true"`. You may even import classes and interfaces that are not explicitly mapped.

Note : Hibernate Types

Values are primitives, collections ,components and certain immutable objects.
Unlike entities ,values (in particular collections and components) *are* persisted and deleted by reachability. Since value objects (and primitives) are persisted and deleted along with their containing entity they may not be independently versioned.
Values have no independent identity, so they cannot be shared by two entities or collections.

-A *component* is a user defined class with value semantics.
- user-defined types may be mapped with entity or value type semantics.
- All built-in Hibernate types except collections support null semantics.

- The basic value types have corresponding `Type` constants defined on `org.hibernate.Hibernate`.

Note : Custom value types

To implement a custom type, implement either `org.hibernate.UserType` or `org.hibernate.CompositeUserType` and declare properties using the fully qualified classname of the type.

- The `CompositeUserType`, `EnhancedUserType`, `UserCollectionType`, and `UserVersionType` interfaces provide support for more specialized uses.

- You may even supply parameters to a `UserType` in the mapping file. To do this, your `UserType` must implement the `org.hibernate.usertype.ParameterizedType` interface. To supply parameters to your custom type, you can use the `<type>` element in your mapping files.

```
<property name="priority">
    <type name="com.mycompany.usertypes.DefaultValueIntegerType">
        <param name="default">0</param>
    </type>
</property>
```

Note : Auxiliary Database Objects
Allows CREATE and DROP of arbitrary database objects, to provide the ability to fully define a user schema within the Hibernate mapping files.

Although designed specifically for creating and dropping things like triggers or stored procedures, really any SQL command that can be run via a `java.sql.Statement.execute()` method is valid here

Note:Automatic Dirty Checking:
-Hibernate automatically detects object state changes in order to synchronize the updated state with the database. It's usually completely safe to return a different object from the getter method to the object passed by Hibernate to the setter. Hibernate will compare the **objects by value**—not by **object identity**—to determine if the property's persistent state needs to be updated. But **Collections** are compared **by identity.**

-For a property mapped as a persistent **collection,** you should return *exactly* the same collection instance from the getter method as Hibernate passed to the setter method. If you don't, Hibernate will update the database, even if no update is necessary, *every time* the session synchronizes state held in memory with the database.

-When you use Load() method ,hibernate automatically detects that the collections has been modified and need to be updated .This is called Automatic Dirty Checking.

Note:
-*Object identity*, ==, is a notion defined by the Java virtual machine. Two object references are identical if they point to the same memory location.

-*object equality* is a notion defined by classes that implement the *`equals()`* method, sometimes also referred to as *equivalence*. Equivalence means that two different (non-identical) objects have the same value.

A persistent object is an in-memory representation of a particular row of a database table.

There three methods for identifying objects:
▪ *Object identity*—Objects are identical if they occupy the same memory location in the JVM. This can be checked by using the == operator.

■ *Object equality*—Objects are equal if they have the same value, as defined by the `equals(Object o)` method. Classes that don't explicitly override this method inherit the implementation defined by `java.lang.Object`, which compares object identity.

■ *Database identity*—Objects stored in a relational database are identical if they represent the same row or, equivalently, share the same table and primary key value.

Database identity with Hibernate
Hibernate exposes database identity to the application in two ways:
- ■ The value of the *identifier property* of a persistent instance
- ■ The value returned by `Session.getIdentifier(Object o)`

Note:
The *candidate key* is a column or set of columns that uniquely identifies a specific row of the table. A candidate key must satisfy the following properties:
■ The value or values are never null.
■ Each row has a unique value or values.
■ The value or values of a particular row never change.

Note:
-Surrogate keys have no business meaning—they are unique values generated by the database or application. There are a number of well-known approaches to surrogate key generation.
-you may create your own identifier generator by implementing Hibernate's *IdentifierGenerator* interface.

Note:
Hibernate makes the following essential distinction:
■ An object of ***entity*** **type** has its own database identity (primary key value). An object reference to an entity is persisted as a reference in the database (a foreign key value). An entity has its own lifecycle; it may exist independently of any other entity.

■ An object of ***value type*** has no database identity; it belongs to an entity, and its persistent state is embedded in the table row of the owning entity (**except in the case of collections, which are also considered value types**). Value types don't have identifiers or identifier properties.The lifespan of a value-type instance is bounded by the lifespan of the owning entity.The most obvious value types are simple objects like `Strings` and `Integers`.

Note:
-Hibernate also lets you treat a user-defined class as a **value type**
-A component has no identity, hence the persistent component class requires no identifier property or identifier mapping.
-Hibernate uses the term *component* for a user-defined class that is persisted to the same table as the owning entity
-Hibernate supports both unidirectional and bidirectional compositions; however, unidirectional composition is far more common.

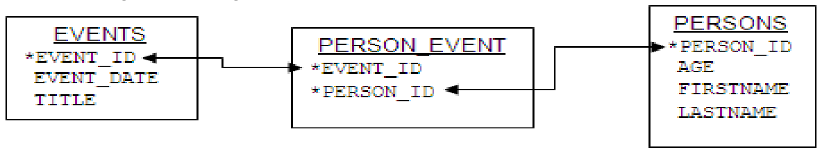
Note:
-There are two important limitations to classes mapped as components:
■ Shared references aren't possible.
■ There is no elegant way to represent a null reference to an attribute.

Note:
Hibernate represents null components as null values in all mapped columns of the component. This means that if you store a component object with all null property values, Hibernate will return a null component when the owning entity object is retrieved from the database.

Note :Disadvantage of Assgned generator class
Detached objects and transitive persistence

Note:
-If an association is bidirectional, both sides of the relationship must be considered.

- For *many to many* association ,an association table is needed.



-All Bi-directional associations need one side as **inverse.**
a. In *one to many* associations it has to be *on the many* side.
b. In *many to many* associations you can pick either side, there is no difference.

***flush*, occurs by default at the following points**
• before some query executions
• from `org.hibernate.Transaction.commit()`
• from `Session.flush()`

The SQL statements are issued in the following order
1. all entity insertions, in the same order the corresponding objects were saved using Session.save()
2. all entity updates
3. all collection deletions
4. all collection element deletions, updates and insertions
5. all collection insertions
6. all entity deletions, in the same order the corresponding objects were deleted using `Session.delete()`

Except when you explicity `flush()`, there are absolutely no guarantees about *when* the `Session` executes the JDBC calls, only the *order* in which they are executed. However, Hibernate does guarantee that the `Query.list(..)` will never return stale data; nor will they return the wrong data.

It is possible to change the default behavior so that flush occurs less frequently. The `FlushMode` class defines three different modes: only flush at commit time (and only when the Hibernate `Transaction` API is used), flush automatically using the explained routine, or never flush unless `flush()` is called explicitly. The last mode is useful for long running units of work, where a `Session` is kept open and disconnected for a long time.

During flush, an exception might occur (e.g. if a DML operation violates a constraint).

Usually, ending a `Session` involves four distinct phases:
• flush the session
• commit the transaction
• close the session
• handle exceptions

Query Language Substitution:
Example 1:
hibernate.query.substitutions true=1, false=0

Example 2:
hibernate.query.substitutions toLowercase=LOWER

Note: Filter
- A *Hibernate filter* is a global, named, parameterized filter that may be enabled or disabled for a particular Hibernate session.Filters can be used like database views, but parameterized inside the application.

- The methods on Session are: enableFilter(String filterName), getEnabledFilter(String filterName), and disableFilter(String filterName). By default, filters are *not* enabled for a given session; they must be explcitly enabled through use of the Session.enabledFilter() method, which returns an instance of the Filter interface.

- Methods on the org.hibernate.Filter interface do allow the method-chaining common to much of Hibernate.

- If you plan on using filters with outer joining (either through HQL or load fetching) be careful of the direction of the condition expression. Its safest to set this up for left outer joining; in general, place the parameter first followed by the column name(s) after the operator.

-Collection filters have an implicit from clause and an implicit where condition.

You can use it to paginate collection elements:

```
List results = session.createFilter( item.getBids(), "" )
.setFirstResult(50)
.setMaxResults(100)
.list();
```

-The most important reason for the existence of collection filters is to allow the application to retrieve some elements of a collection without initializing the entire collection.

Filtering collections

-A collection *filter* is a special type of query that may be applied to a persistent collection or array. The query string may refer to `this`, meaning the current collection element.

-The returned collection is considered a bag, and it's a copy of the given collection. The original collection is not modified.

-filters do not require a `from` clause (though they may have one if required). Filters are not limited to returning the collection elements themselves.

-Even an empty filter query is useful, e.g. to load a subset of elements in a huge collection.

Note :

Hibernate supports an easy-to-use but powerful object oriented query language (HQL). For programmatic query creation, Hibernate supports a sophisticated Criteria and Example query feature (QBC and QBE).

HQL supports the following:
■ The ability to apply restrictions to properties of associated objects related by reference or held in collections (to navigate the object graph using query language).
■ The ability to retrieve only properties of an entity or entities, without the overhead of loading the entity itself in a transactional scope. This is sometimes called a *report query*; it's more correctly called *projection*.
■ The ability to order the results of the query.
■ The ability to paginate the results.
■ Aggregation with group by, having, and aggregate functions like sum, min,and max.
■ Outer joins when retrieving multiple objects per row.
■ The ability to call user-defined SQL functions.
■ Subqueries (nested queries).

-Along with `elements()`, HQL provides `indices()`, `maxelement()`, `minelement()`,`maxindex()`, `minindex()`, and `size()`, each of which is equivalent to a certain correlated subquery against the passed collection.

-HQL provides no mechanism for specifying SQL query hints, and it also doesn't support hierarchical queries (such as the Oracle `CONNECT BY` clause).

Note (HQL)

-It is considered good practice to name query aliases using an initial lowercase, consistent with Java naming standards for local variables (eg. domesticCat).

-The supported join types are borrowed from ANSI SQL
• inner join
• left outer join
• right outer join
• full join (not usually useful)

HQL provides four ways of expressing (inner and outer) joins:
■ An *ordinary* join in the from clause
■ A *fetch* join in the from clause
■ A *theta-style* join in the where clause
■ An *implicit* association join

-we prefer to map all associations lazy by default, so an eager, outerjoin fetch query is used to override the default fetching strategy at runtime.

Note : Points to remember
■ *HQL always ignores the mapping document eager fetch (outer join) setting.*
If you've mapped some associations to be fetched by outer join (by setting outer-join="true" on the association mapping), any HQL query will ignore this preference. You must use an explicit fetch join if you want eager fetching in HQL. On the other hand, the criteria query will not ignore the mapping! If you specify outer-join="true" in the mapping file, the criteriaquery will fetch that association by outer join—just like Session.get() or Session.load() for retrieval by identifier. For a criteria query, you can explicitly disable outer join fetching by calling setFetchMode("bids",FetchMode.LAZY). HQL is designed to be as flexible as possible: You can completely (re)define the fetching strategy that should be used at runtime.

■ *Hibernate currently limits you to fetching just one collection eagerly.*
This is a reasonable restriction, since fetching more than one collection in a single query would be a Cartesian product result.

■ *If you fetch a collection, Hibernate doesn't return a distinct result list.*

-by default, HQL queries return all queried entities if we don't select explicitly.

-In Hibernate, you can use an *implicit association join*.

-You may supply extra join conditions using the HQL `with` keyword.

```
from Cat as cat
    left join cat.kittens as kitten
        with kitten.bodyWeight > 10.0
```

-a "fetch" join allows associations or collections of values to be initialized along with their parent objects, using a single select. This is particularly useful in the case of a collection. It effectively overrides the outer join and lazy declarations of the mapping file for associations and collections.

-Note that the `fetch` construct may not be used in queries called using `scroll()` or `iterate()`. Nor should `fetch` be used together with `setMaxResults()` or `setFirstResult()`. Nor may `fetch` be used together with an ad hoc `with` condition.

-`full join fetch` and `right join fetch` are not meaningful.

-If you are using property-level lazy fetching, it is possible to force Hibernate to fetch the lazy properties immediately (in the first query) using `fetch all properties`.

Forms of join syntax

-HQL supports two forms of association joining: `implicit` and `explicit`.

-Queries may return multiple objects and/or properties as an array of type `Object[]`,

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

-or as a `List`,

```
select new list(mother, offspr, mate.name)
from DomesticCat as mother
    inner join mother.mate as mate
    left outer join mother.kittens as offspr
```

-or as an actual typesafe Java object,

```
select new Family(mother, mate, offspr)
from DomesticCat as mother
    join mother.mate as mate
    left join mother.kittens as offspr
```

-You may assign aliases to selected expressions using `as`:

```
select new map( max(bodyWeight) as max, min(bodyWeight) as min, count(*) as n )
from Cat cat
```

Note:

-By default, a criteria query returns only the root entity..

```
Iterator items =
session.createCriteria(Item.class)
.createAlias("bids", "bid")
.add( Expression.like("this.description", "%gc%") )
.add( Expression.gt("bid.amount", new BigDecimal("100") ) )
.list().iterator();
while ( items.hasNext() ) {
Item item = (Item) items.next();
// Do something
}
```

-A limitation of criteria queries is that you can't combine a createAlias with an eager fetch mode;

If we want to return both the matching Items and Bids, we must ask Hibernate to return each row of results as a Map:

```
Iterator itemBidMaps =
session.createCriteria(Item.class)
.createAlias("bids", "bid")
.add( Expression.like("this.description", "%gc%") )
.add( Expression.gt("bid.amount", new BigDecimal("100") ) )
.returnMaps()
.list().iterator();
while ( itemBidMaps.hasNext() ) {
Map map = (Map) itemBidMaps.next();
Item item = (Item) map.get("this");
Bid bid = (Bid) map.get("bid");
// Do something
}
```

-Note that the Criteria API doesn't provide any means for expressing Cartesian products or theta-style joins. It's also currently not possible in Hibernate to outer-join two tables that don't have a mapped association.

Note:
-If you chose to use the Hibernate API, you must enclose SQL aliases in braces.

-For Native SQL no need of hbm file to be present because we directly use relative database specific queries.

-For selecting all the columns then HQL (or) Criteria is better.[here hbm file is required]

-For selecting only few columns then NativeSQL is best. [here hbm file is not required].

-addScalar() is used to specify the only Datatype.

-addEntity() is used to specify the complete object, so as to map all the results to a java object.

-If queries are simple then no need to write in hbm file, we will write in java file.

-If queries involves UNIONs and more complex then we need to move the query to the hbm file.

-In hbm file we have to write the NativeSQL after </class> tag.

-Named SQL query is in the hbm file [HQL or Native SQL],no need to use addEntity() in this case.

-Since the native SQL is tightly coupled to the actual mapped tables and columns,we strongly recommend that you define all native SQL queries in the mapping document instead of embedding them in the Java code.

Note:
-The stored procedure/function must return a resultset as the first out-parameter to be able to work with Hibernate. To use this query in Hibernate you need to map it via a named query.

- If you still want to use these procedures you have to execute them via `session.connection()`. The rules are different for each database, since database vendors have different stored procedure semantics/syntax.

Rules/limitations for using stored procedures

1. Stored procedure queries can't be paged with `setFirstResult()/setMaxResults()`.
2. Recommended call form is standard SQL92:
   `{ ? = call functionName(<parameters>) }` or `{ ? = call procedureName(<parameters>}`.
   Native call syntax is not supported.

For Oracle the following rules apply:

• A function must return a result set. The first parameter of a procedure must be an `OUT` that returns a resultset.

Note:- Stored procedures are supported if the `callable` attribute is set:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <sql-insert callable="true">{call createPerson (?, ?)}</sql-insert>
    <sql-delete callable="true">{? = call deletePerson (?)}</sql-delete>
    <sql-update callable="true">{? = call updatePerson (?, ?)}</sql-update>
</class>
```

- You can see the expected order by enabling debug logging for the `org.hibernate.persister.entity` level. With this level enabled Hibernate will print out the static SQL that is used to create, update, delete etc. entities.
- The stored procedures are in most cases.Hibernate always registers the first statement parameter as a numeric output parameter for the CUD operations.

Externalizing named queries

```
<query name="eg.DomesticCat.by.name.and.minimum.weight">
      <![CDATA[from eg.DomesticCat as cat where cat.name = ? and cat.weight > ? ] ]>
</query>

Query q = sess.getNamedQuery("eg.DomesticCat.by.name.and.minimum.weight");
q.setString(0, name);
q.setInt(1, minWeight);
List cats = q.list();
```

Note that the actual program code is independent of the query language that is used, you may also define native SQL queries in metadata, or migrate existing queries to Hibernate by placing them in mapping files.

Note: Custom SQL for loading

```
<sql-query name="person">
    <return alias="pers" class="Person" lock-mode="upgrade"/>
    SELECT NAME AS {pers.name}, ID AS {pers.id}
    FROM PERSON
    WHERE ID=?
    FOR UPDATE
</sql-query>
```

This is just a named query declaration, as discussed earlier. You may reference this named query in a class mapping:

```
<class name="Person">
    <id name="id">
        <generator class="increment"/>
    </id>
    <property name="name" not-null="true"/>
    <loader query-ref="person"/>
</class>
```

This even works with stored procedures.
You may even define a query for collection loading:

```
<set name="employments" inverse="true">
    <key/>
    <one-to-many class="Employment"/>
    <loader query-ref="employments"/>
</set>
```

```
<sql-query name="employments">
    <load-collection alias="emp" role="Person.employments"/>
    SELECT {emp.*}
    FROM EMPLOYMENT emp
    WHERE EMPLOYER = :id
    ORDER BY STARTDATE ASC, EMPLOYEE ASC
</sql-query>
```

You could even define an entity loader that loads a collection by join fetching:

```
<sql-query name="person">
    <return alias="pers" class="Person"/>
    <return-join alias="emp" property="pers.employments"/>
    SELECT NAME AS {pers.*}, {emp.*}
    FROM PERSON pers
    LEFT OUTER JOIN EMPLOYMENT emp
        ON pers.ID = emp.PERSON_ID
    WHERE ID=?
</sql-query>
```

The fetch strategy defined in the mapping document affects:
• retrieval via get() or load()
• retrieval that happens implicitly when an association is navigated
• Criteria queries
• HQL queries if subselect fetching is used

-using `left join fetch` in HQL. This tells Hibernate to fetch the association eagerly in the first select, using an outer join. In the `Criteria` query API, you would use `setFetchMode(FetchMode.JOIN)`.

-By contrast, a criteria query defines an implicit alias. The root entity in a criteria query is always assigned the alias `this`.

Improving performance

Fetching strategies
A *fetching strategy* is the strategy Hibernate will use for retrieving associated objects if the application needs to navigate the association. Fetch strategies may be declared in the O/R mapping metadata, or over-ridden by a particular HQL or Criteria query.

Hibernate3 defines the following fetching strategies:
• *Join fetching* - Hibernate retrieves the associated instance or collection in the same SELECT, using an OUTER JOIN.

• *Select fetching* - a second SELECT is used to retrieve the associated entity or collection. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you actually access the association.

• *Subselect fetching* - a second SELECT is used to retrieve the associated collections for all entities retrieved in a previous query or fetch. Unless you explicitly disable lazy fetching by specifying lazy="false", this second select will only be executed when you actually access the association.

• *Batch fetching* - an optimization strategy for select fetching - Hibernate retrieves a batch of entity instances or collections in a single SELECT, by specifying a list of primary keys or foreign keys.

Hibernate also distinguishes between:
• *Immediate fetching* - an association, collection or attribute is fetched immediately, when the owner is loaded.

• *Lazy collection fetching* - a collection is fetched when the application invokes an operation upon that collection. (This is the default for collections.)

• *"Extra-lazy" collection fetching* - individual elements of the collection are accessed from the database as needed. Hibernate tries not to fetch the whole collection into memory unless absolutely needed (suitable for very large collections)

• *Proxy fetching* - a single-valued association is fetched when a method other than the identifier getter is invoked upon the associated object.

• *"No-proxy" fetching* - a single-valued association is fetched when the instance variable is accessed. Compared to proxy fetching, this approach is less lazy (the association is fetched even when only the identifier is accessed) but more transparent, since no proxy is visible to the application. This approach requires buildtime bytecode instrumentation and is rarely necessary.

• *Lazy attribute fetching* - an attribute or single valued association is fetched when the instance variable is accessed.This approach requires buildtime bytecode instrumentation and is rarely necessary.

Hibernate allows you to choose among four fetching strategies for any association, in association metadata and at runtime:
■ *Immediate fetching*—The associated object is fetched immediately, using a sequential database read (or cache lookup).

■ *Lazy fetching*—The associated object or collection is fetched "lazily," when it's first accessed. This results in a new request to the database (unless the associated object is cached).

■ *Eager fetching*—The associated object or collection is fetched together with the owning object, using an SQL outer join, and no further database request is required.

■ *Batch fetching*—This approach may be used to improve the performance of lazy fetching by retrieving a batch of objects or collections when a lazy association is accessed. (Batch fetching may also be used to improve the performance of immediate fetching.)

Note:
-By default, Hibernate3 uses lazy select fetching for collections and lazy proxy fetching for single-valued associations.
These defaults make sense for almost all associations in almost all applications.

-Note: if you set hibernate.default_batch_fetch_size, Hibernate will use the batch fetch optimization for lazy fetching (this optimization may also be enabled at a more granular level).
However, lazy fetching poses one problem that you must be aware of. Access to a lazy association outside of the context of an open Hibernate session will result in an exception.

-Hibernate does not support lazy initialization for detached object

Outer Join Fetching:
If your database supports ANSI, Oracle or Sybase style outer joins, *outer join fetching* will often increase performance by limiting the number of round trips to and from the database . Outer join fetching allows a whole graph of objects connected by many-to-one,one-to-many, many-to-many and one-to-one associations to be retrieved in a single SQL SELECT.

Outer join fetching may be disabled *globally* by setting the property hibernate.max_fetch_depth to 0. A setting of 1 or higher enables outer join fetching for one-to-one and many-to-one associations which have been mapped with fetch="join".

Note: Mapping class inheritance

▪ ***Table per concrete class***—Discard polymorphism and inheritance relationships completely from the relational model. `<class>`

Disadvantages :
1. It doesn't support polymorphic associations very well.
2. *Polymorphic queries* are also problematic. {A separate query is needed for each concrete subclass.}
3. This makes schema evolution more complex.
4. It also makes it much more difficult to implement database integrity constraints that apply to all subclasses.

Note: We recommend this approach (only) for the top level of your class hierarchy, where polymorphism isn't usually required.

▪ ***Table per class hierarchy***—Enable polymorphism by denormalizing the relational model and using a type discriminator column to hold type information. Hibernate will automatically set and retrieve the discriminator values. The `<subclass>` element can in turn contain other `<subclass>` elements, until the whole hierarchy is mapped to the table. A `<subclass>` element can't contain a `<joined-subclass>` element.

Advantages :
- Both polymorphic and nonpolymorphic queries perform well.
- it's even easy to implement by hand.
- Ad hoc reporting is possible without complex joins or unions, and schema evolution is straightforward.

Disadvantage:
- Columns for properties declared by subclasses must be declared to be nullable.

■ ***Table per subclass***—Represent "is a" (inheritance) relationships as "has a" (foreign key) relationships.To represent inheritance relationships as relational foreign key associations Schema evolution and integrity constraint definition are straightforward. A polymorphic association to a particular subclass may be represented as a foreign key pointing to the table of that subclass. In Hibernate, we use the `<joined-subclass>` element to indicate a table-per-subclass mapping. No discriminator is required with this strategy.

For ad hoc reporting, database views provide a way to offset the complexity of the table-per-subclass strategy. A view may be used to transform the table-per-subclass model into the much simpler table-per-hierarchy model. Hibernate will use an **outer join** when querying the **base class** and an **inner join** when querying the **subclass.**

Advantages:
-Relational model is completely normalized.

Disadvantage:
- This mapping strategy is more difficult to implement by hand—
-Ad hoc reporting will be more complex.

Note: *Choosing a strategy*
- You can apply all mapping strategies to abstract classes and interfaces. you can map any declared or inherited property using `<property>`. Hibernate won't try to instantiate an abstract class, however, even if you query or load it.

Here are some rules of thumb:
-If you don't require polymorphic associations or queries, lean toward the table-per-concrete-class strategy.

-If you require polymorphic associations or queries, and subclasses declare relatively few properties lean toward the table-per-class-hierarchy model.

- If you require polymorphic associations or queries, and subclasses declare many properties lean toward the table-per-subclass approach.

By default, choose table-per-class-hierarchy for simple problems.
For more complex cases, you should consider the table-per-subclass strategy.

<u>**Note:**</u>
Polymorphic queries : queries that return objects of all classes that match the interface of the queried class

Features of inheritance mappings

Inheritance strategy	**Polymorphic Many-to-one**	**Polymorphic one-to-one**	**Polymorphic One-to-many**	**Polymorphic Many-to-many**	**Polymorphic load()/get()**	**Polymorphic queries**	**Polymorphic joins**
table per class hierarchy	<many-to-one>	<one-to-one>	<one-to-many>	<many-to-many>	s.get(Payment.class, id)	from Payment p	from Order o join o.payment p
table per subclass	<many-to-one>	<one-to-one>	<one-to-many>	<many-to-many>	s.get(Payment.class, id)	from Payment p	from Order o join o.payment p
table per concreteclass (union-subclass)	<many-to-one>	<one-to-one>	<one-to-many> (for inverse="true" only)	<many-to-many>	s.get(Payment.class, id)	from Payment p	from Order o join o.payment p
table per concrete class (implicit polymorphism)	<any>	*not supported*	*not supported*	<many-to-many>	s.createCriteria(Payment.class).add(Restrictions. idEq(id)) .uniqueResult()	from Payment p	*not supported*

Tips & Tricks

1.You can count the number of query results without actually returning them:

```
((Integer) session.iterate("select count(*) from ....").next() ).intValue()
```

2.To order a result by the size of a collection, use the following query:

```
select usr.id, usr.name
from User as usr
left join usr.messages as msg
group by usr.id, usr.name
order by count(msg)
```

3.If your database supports subselects, you can place a condition upon selection size in the where clause of your query:

```
from User usr where size(usr.messages) >= 1
```

4.If your database doesn't support subselects, use the following query:

```
select usr.id, usr.name
from User usr.name
join usr.messages msg
group by usr.id, usr.name
having count(msg) >= 1
```

5.As this solution can't return a User with zero messages because of the inner join, the following form is also useful:

```
select usr.id, usr.name
from User as usr
left join usr.messages as msg
group by usr.id, usr.name
having count(msg) = 0
```

6.Properties of a JavaBean can be bound to named query parameters:

```
Query q = s.createQuery("from foo Foo as foo where foo.name=:name and foo.size=:size");
q.setProperties(fooBean); // fooBean has getName() and getSize()
List foos = q.list();
```

7.Collections are pageable by using the Query interface with a filter:

```
Query q = s.createFilter( collection, "" ); // the trivial filter
q.setMaxResults(PAGE_SIZE);
q.setFirstResult(PAGE_SIZE * pageNumber);
List page = q.list();
```

8.Collection elements may be ordered or grouped using a query filter:

```
Collection orderedCollection = s.filter( collection,"order by this.amount" );
Collection counts = s.filter( collection, "select this.type, count(this) group by
this.type" );
```

9.You can find the size of a collection without initializing it:

```
( (Integer) session.iterate("select count(*) from ....").next() ).intValue();
```

Best Practices

-Write fine-grained classes and map them using <component>.
Use an Address class to encapsulate street, suburb, state, postcode. This encourages code reuse and simplifies refactoring.

-Declare identifier properties on persistent classes.
Hibernate makes identifier properties optional. There are all sorts of reasons why you should use them. We recommend that identifiers be 'synthetic' (generated, with no business meaning).

-Identify natural keys.
Identify natural keys for all entities, and map them using <natural-id>. Implement equals() and hash-Code() to compare the properties that make up the natural key.

-Place each class mapping in its own file.
Don't use a single monolithic mapping document. Map com.eg.Foo in the file com/eg/Foo.hbm.xml. This makes particularly good sense in a team environment.

-Load mappings as resources.
Deploy the mappings along with the classes they map.

-Consider externalising query strings.
This is a good practice if your queries call non-ANSI-standard SQL functions. Externalising the query strings to mapping files will make the application more portable.

-Use bind variables.
As in JDBC, always replace non-constant values by "?". Never use string manipulation to bind a non-constant value in a query! Even better, consider using named parameters in queries.

-Don't manage your own JDBC connections.
Hibernate lets the application manage JDBC connections. This approach should be considered a last-resort.If you can't use the built-in connections providers, consider providing your own implementation of org.hibernate.connection.ConnectionProvider.

-Consider using a custom type.
Suppose you have a Java type, say from some library, that needs to be persisted but doesn't provide the accessors needed to map it as a component. You should consider implementing org.hibernate.UserType. This approach frees the application code from implementing transformations to / from a Hibernate type.

-Use hand-coded JDBC in bottlenecks.
In performance-critical areas of the system, some kinds of operations might benefit from direct JDBC. But please, wait until you *know* something is a bottleneck. And don't assume that direct JDBC is necessarily faster. If you need to use direct JDBC, it might be worth opening a Hibernate Session and using that JDBC connection. That way you can still use the same transaction strategy and underlying connection provider.

-Understand Session flushing.
From time to time the Session synchronizes its persistent state with the database. Performance will be affected if this process occurs too often. You may sometimes minimize unnecessary flushing by disabling automatic flushing or even by changing the order of queries and other operations within a particular transaction.

-In a three tiered architecture, consider using detached objects.
When using a servlet / session bean architecture, you could pass persistent objects loaded in the session bean to and from the servlet / JSP layer. Use a new session to service each request. Use Session.merge() or Session.saveOrUpdate() to synchronize objects with the database.

-In a two tiered architecture, consider using long persistence contexts.
Database Transactions have to be as short as possible for best scalability. However, it is often neccessary to implement long running *application transactions*, a single unit-of-work from the point of view of a user.An application transaction might span several client request/response cycles. It is common to use detached objects to implement application transactions. An alternative, extremely appropriate in two tiered architecture, is to maintain a single open persistence contact (session) for the whole lifecycle of the application transaction and simply disconnect from the JDBC connection at the end of each request and reconnect at the beginning of the subsequent request. Never share a single session across more than one application transaction, or you will be working with stale data.

-Don't treat exceptions as recoverable.
This is more of a necessary practice than a "best" practice. When an exception occurs, roll back the Transaction and close the Session. If you don't, Hibernate can't guarantee that in-memory state accurately represents persistent state. As a special case of this, do not use Session.load() to determine if an instance with the given identifier exists on the database; use Session.get() or a query instead.

-Prefer lazy fetching for associations.
Use eager fetching sparingly. Use proxies and lazy collections for most associations to classes that are not likely to be completely held in the second-level cache. For associations to cached classes, where there is an a extremely high probability of a cache hit, explicitly disable eager fetching using lazy="false". When an join fetching is appropriate to a particular use case, use a query with a left join fetch.

-Use the *open session in view* pattern, or a disciplined *assembly phase* to avoid problems with unfetched data. Hibernate frees the developer from writing tedious *Data Transfer Objects* (DTO). In a traditional EJB architecture, DTOs serve dual purposes: first, they work around the problem that entity beans are not serializable; second, they implicitly define an assembly phase where all data to be used by the view is fetched and marshalled into the DTOs before returning control to the presentation tier. Hibernate eliminates the first purpose. However, you will still need an assembly phase (think of your business methods as having a strict contract with the presentation tier about what data is available in the detached objects) unless you are prepared to hold the persistence context (the session) open across the view rendering process. This is not a limitation of Hibernate! It is a fundamental requirement of safe transactional data access.

-Consider abstracting your business logic from Hibernate.
Hide (Hibernate) data-access code behind an interface. Combine the *DAO* and *Thread Local Session* patterns.You can even have some classes persisted by handcoded JDBC, associated to Hibernate via a UserType. (This advice is intended for "sufficiently large" applications; it is not appropriate for an application with five tables!)

-Don't use exotic association mappings.
Good usecases for a real many-to-many associations are rare. Most of the time you need additional information stored in the "link table". In this case, it is much better to use two one-to-many associations to an intermediate link class. In fact, we think that most associations are one-to-many and many-to-one, you should be careful when using any other association style and ask yourself if it is really neccessary.

-Prefer bidirectional associations.
Unidirectional associations are more difficult to query. In a large application, almost all associations must be navigable in both directions in queries.

Tuning object retrieval

1 Enable the Hibernate SQL log,

2 Step through your application use case by use case and note how many and what SQL statements Hibernate executes.

3 You may encounter two common issues:

■ If the SQL statements use join operations that are too complex and slow, set `outer-join` to false for `<many-to-one>` associations (this is enabled by default). Also try to tune with the global `hibernate.max_fetch_depth` configuration option, but keep in mind that this is best left at a value between 1 and 4.

■ If too many SQL statements are executed, use `lazy="true"` for all collection mappings; by default, Hibernate will execute an immediate additional fetch for the collection elements (which, if they're entities, can cascade further into the graph). In rare cases, if you're sure, enable `outer-join="true"` and disable lazy loading for particular collections. Keep in mind that only one collection property per persistent class may be fetched eagerly. Use batch fetching with values between 3 and 10 to further optimize collection fetching if the given unit of work involves several "of the same" collections or if you're accessing a tree of parent and child objects.

4 After you set a new fetching strategy, rerun the use case and check the generated SQL again. Note the SQL statements, and go to the next use case.

5 After you optimize all use cases, check every one again and see if any optimizations had side effects for others. With some experience, you'll be able to avoid any negative effects and get it right the first time.

Note : most common issues of Hibernate.

Optimizing object retrieval

a) Solving the n+1 selects problem

-The biggest performance killer in applications that persist objects to SQL databases is the *n+1 selects problem.*It's normal (and recommended) to map almost all associations for lazy initialization.This means you generally set all collections to lazy="true" and even change some of the one-to-one and many-to-one associations to not use outer joins by default. This is the only way to avoid retrieving all objects in the database in every transaction. Unfortunately, this decision exposes you to the n+1 selects problem. It's easy to understand this problem by considering a simple query that retrieves all Items for a particular user:

```
Iterator items = session.createCriteria(Item.class)
.add( Expression.eq("item.seller", user) )
.list()
.iterator();
```

This query returns a list of items, where each collection of bids is an uninitialized collection wrapper. Suppose that we now wish to find the maximum bid for each item. The following code would be one way to do this:

```
List maxAmounts = new ArrayList();
while (items.hasNext()) {
Item item = (Item) items.next();
BigDecimal maxAmount = new BigDecimal("0");
for ( Iterator b = item.getBids().iterator(); b.hasNext(); ) {
Bid bid = (Bid) b.next();
if ( bid.getAmount().compareTo(maxAmount) == 1 )
maxAmount = bid.getAmount();
}
maxAmounts.add( new MaxAmount( item.getId(), maxAmount ) );
}
```

But there is a huge problem with this solution (aside from the fact that this would be much better executed in the database using aggregation functions): Each time we access the collection of bids, Hibernate must fetch this lazy collection from the database for each item. If the initial query returns 20 items, the entire transaction requires 1 initial select that retrieves the items plus 20 additional selects to load the bids collections of each item. This might easily result in unacceptable latency in a system that accesses the database across a network. Usually you don't explicitly create such operations, because you should quickly see doing so is suboptimal. However, the n+1 selects problem is often hidden in more complex application logic, and you may not recognize it by looking at a single routine.

The first attempt to solve this problem might be to enable *batch fetching*. We change our mapping for the bids collection to look like this:

```
<set name="bids" lazy="true" inverse="true" batch-size="10">
```

With batch fetching enabled, Hibernate prefetches the next 10 collections when the first collection is accessed. This reduces the problem from n+1 selects to n/10+ 1 selects. For many applications, this may be sufficient to achieve acceptable latency. On the other hand, it also means that in some other transactions, collections are fetched unnecessarily. It isn't the best we can do in terms of reducing the number of round trips to the database.

A much, much better solution is to take advantage of HQL aggregation and perform the work of calculating the maximum bid on the database. Thus we avoid the problem:

```
String query = "select MaxAmount( item.id, max(bid.amount) )"
+ " from Item item join item.bids bid"
+ " where item.seller = :user group by item.id";
List maxAmounts = session.createQuery(query)
.setEntity("user", user)
.list();
```

Unfortunately, this isn't a complete solution to the generic issue. In general, we may need to do more complex processing on the bids than merely calculating the maximum amount. We'd prefer to do this processing in the Java application.
We can try enabling eager fetching at the level of the mapping document:

```
<set name="bids" inverse="true" outer-join="true">
```

The outer-join attribute is available for collections and other associations. It forces Hibernate to load the association eagerly, using an SQL outer join.
we consider eager fetching at the level of the mapping file to be almost always a bad approach. The outer-join attribute of collection mappings is arguably a misfeature of Hibernate (fortunately, it's disabled by default). Occasionally it makes sense to enable outer-join for a <many-to-one> or <one-to-one> association (the default is auto; see chapter 4, section 4.4.6.1, "Single point associations"), but we'd never do this in the case of a collection.

Our recommended solution for this problem is to take advantage of Hibernate's support for runtime (code-level) declarations of association fetching strategies. The example can be implemented like this:

```
List results = session.createCriteria(Item.class)
.add( Expression.eq("item.seller", user) )
.setFetchMode("bids", FetchMode.EAGER).list();
// Make results distinct
Iterator items = new HashSet(results).iterator();
List maxAmounts = new ArrayList();
for ( ; items.hasNext(); ) {
Item item = (Item) items.next();
BigDecimal maxAmount = new BigDecimal("0");
for ( Iterator b = item.getBids().iterator(); b.hasNext(); ) {
Bid bid = (Bid) b.next();
if ( bid.getAmount().compareTo(maxAmount) == 1 )
maxAmount = bid.getAmount();
}
maxAmounts.add( new MaxAmount( item.getId(), maxAmount ) );
}
```

We disabled batch fetching and eager fetching at the mapping level; the collection is lazy by default. Instead, we enable eager fetching for this query alone by calling setFetchMode(). As discussed earlier in this chapter, this is equivalent to a fetch
join in the from clause of an HQL query.

The previous code example has one extra complication: The result list returned by the Hibernate criteria query isn't guaranteed to be distinct. In the case of a query that fetches a collection by outer join, it will contain duplicate items. It's the application's responsibility to make the results distinct if that is required. We implement this by adding the results to a HashSet and then iterating the set.

So, we have established a general solution to the n+1 selects problem. Rather than retrieving just the top-level objects in the initial query and then fetching needed associations as the application navigates the object graph, we follow a two step process:
1 Fetch all needed data in the initial query by specifying exactly which associations will be accessed in the following unit of work.
2 Navigate the object graph, which will consist entirely of objects that have already been fetched from the database.

This is the only true solution to the mismatch between the object-oriented world, where data is accessed by navigation, and the relational world, where data is accessed by joining.

Finally, there is one further solution to the n+1 selects problem. For some classes or collections with a sufficiently small number of instances, it's possible to keep all instances in the second-level cache, avoiding the need for database access.
Obviously, this solution is preferred where and when it's possible (it isn't possible in the case of the bids of an Item, because we wouldn't enable caching for this kind of data).

The n+1 selects problem may appear whenever we use the list() method of Query to retrieve the result. As we mentioned earlier, this issue can be hidden in more complex logic; we highly recommend the optimization strategies mentioned
in chapter 4, section 4.4.7, "Tuning object retrieval" to find such scenarios.

It's also possible to generate too many selects by using find(), the shortcut for queries on the Session API, or load() and get().

There is a third query API method we haven't discussed yet. It's extremely important to understand when it's applicable, because it produces n+1 selects!

7.6.2 Using iterate() queries
The iterate() method of the Session and Query interfaces behaves differently than the find() and list() methods. It's provided specifically to let you take full advantage of the second-level cache.
Consider the following code:

```
Query categoryByName =
session.createQuery("from Category c where c.name like :name");
categoryByName.setString("name", categoryNamePattern);
List categories = categoryByName.list();
```

This query results in execution of an SQL select, with all columns of the CATEGORY table included in the select clause:

```
select CATEGORY_ID, NAME, PARENT_ID from CATEGORY where NAME like ?
```

If we expect that categories are already cached in the session or second-level cache, then we only need the identifier value (the key to the cache). This will reduce the amount of data we have to fetch from the database. The following SQL would be slightly more efficient:

```
select CATEGORY_ID from CATEGORY where NAME like ?
```

We can use the `iterate()` method:

```
Query categoryByName =
session.createQuery("from Category c where c.name like :name");
categoryByName.setString("name", categoryNamePattern);
Iterator categories = categoryByName.iterate();
```

The initial query only retrieves the category primary key values. We then iterate through the result, and Hibernate looks up each Category in the current session and in the second-level cache. If a cache miss occurs, Hibernate executes an additional select, retrieving the category by its primary key from the database.

In most cases, this is a minor optimization. It's usually much more important to minimize *row* reads than to minimize *column* reads. Still, if your object has large string fields, this technique may be useful to minimize data packets on the network and, therefore, latency.

Let's talk about another optimization, which also isn't applicable in every case.So far, we've only discussed caching the results of a lookup by identifier (including implicit lookups, such as loading a lazy association) in chapter 5. It's also possible to cache the results of Hibernate queries.

7.6.3 Caching queries

For applications that perform many queries and few inserts, deletes, or updates, caching queries can have an impact on performance. However, if the application performs many writes, the query cache won't be utilized efficiently. Hibernate expires a cached query result set when there is *any* insert, update, or delete of any row of a table that appears in the query.

Just as not all classes or collections should be cached, not all queries should be cached or will benefit from caching. For example, if a search screen has many different search criteria, then it's unlikely that the user will choose the same criterion twice. In this case, the cached query results won't be utilized, and we'd be better off not enabling caching for that query.

Note that the query cache does *not* cache the entities returned in the query result set, just the identifier values**.** Hibernate will, however, fully cache the value typed data returned by a projection query.

For example, the projection query "`select u, b.created from User u, Bid b where b.bidder = u`" will result in caching of the identifiers of the users and the date object when they made their bids. It's the responsibility of the second-level cache (in conjunction with the session cache) to cache the actual state of entities. So, if the cached query you just saw is executed again, Hibernate will have the bid-creation dates in the query cache but perform a lookup in the session and second-level cache (or even execute SQL again) for each user that was in the result. This is similar to the lookup strategy of iterate(), as explained in the previous section.

The query cache must be enabled using a Hibernate property setting:

```
hibernate.cache.use_query_cache true
```

However, this setting alone isn't enough for Hibernate to cache query results. By default, Hibernate queries always ignore the cache. To enable query caching for a particular query (to allow its results to be added to the cache, and to allow it to draw its results *from* the cache), you use the Query interface:

```
Query categoryByName =
session.createQuery("from Category c where c.name = :name");
categoryByName.setString("name", categoryName);
categoryByName.setCacheable(true);
```

Even this doesn't give you sufficient granularity, however. Different queries may require different query expiration policies. Hibernate allows you to specify a different named cache *region* for each query:

```
Query userByName =
session.createQuery("from User u where u.username= :uname");
userByName.setString("uname", username);
userByName.setCacheable(true);
userByName.setCacheRegion("UserQueries");
```

You can now configure the cache expiration policies using the region name. When query caching is enabled, the cache regions are as follows:
■ The default query cache region, `net.sf.hibernate.cache.QueryCache`
■ Each named region
▪ The *timestamp cache*, `net.sf.hibernate.cache.UpdateTimestampsCache`, which is a special region that holds timestamps of the most recent updates to each table

Hibernate uses the timestamp cache to decide if a cached query result set is stale. Hibernate looks in the timestamp cache for the timestamp of the most recent insert, update, or delete made to the queried table. If it's later than the timestamp of the cached query results, then the cached results are discarded and a new query is issued. For best results, you should configure the timestamp cache so that the update timestamp for a table doesn't expire from the cache while queries against the table are still cached in one of the other regions. The easiest way is to turn off expiry for the timestamp cache.

Some final words about performance optimization: Remember that issues like the n+1 selects problem can slow your application to unacceptable performance.

Try to avoid the problem by using the best fetching strategy. Verify that your object-retrieval technique is the best for your use case before you look into caching anything.

From our point of view, caching at the second level is an important feature, but it isn't the first option when optimizing performance. Errors in the design of queries or an unnecessarily complex part of your object model can't be improved with a "cache it all" approach. If an application only performs at an acceptable level with a
hot cache (a full cache) after several hours or days of runtime, you should check it for serious design mistakes, unperformant queries, and n+1 selects problems.

Note:
automatic dirty checking: This feature saves us the effort of explicitly asking Hibernate to update the database when we modify the state of an object inside a transaction.

cascading save: It saves us the effort of explicitly making the new object persistent by calling save(), as long as it's reachable by an already persistent instance.

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.* The SQL statements are usually issued at the end of a transaction. This behavior is called *write-behind,*

Note :
The advantages of named parameters are:
• named parameters are insensitive to the order they occur in the query string
• they may occur multiple times in the same query
• they are self-documenting

Note :

There are two common strategies when dealing with updates to database records ,pessimistic locking and optimistic locking.

Optimistic locking is more scalable than pessimistic locking when dealing with a highly concurrent environment. However pessimistic locking is a better solution for situations where the possibility of simultaneous updates to the same data by multiple sources (for example, users) is common, hence making the possibility of "data clobbering," a likely scenario. Let's look at a brief explanation of each of these two locking strategies.

Pessimistic locking is when you want to reserve a record for exclusive update by locking the database record (or entire table). Hibernate supports pessimistic locking (using the underlying database, not in-memory) via one of the following methods:

- `Session.get`
- `Session.load`
- `Session.lock`
- `Session.refresh`
- `Query.setLockMode`

Optimistic locking means that you will not lock a given database record or table and instead check a column/property of some sort (for example, a timestamp column) to ensure the data has not changed since you read it. Hibernate supports this using a `version` property, which can either be checked manually by the application or automatically by Hibernate for a given session.

Advantages of Hibernate

- **Caching objects.** The session is a transaction-level cache of persistent objects. You may also enable a JVM-level/cluster cache to memory and/or local disk.

- **Executing SQL statements later,** when needed. The session never issues an INSERT or UPDATE until it is actually needed. So if an exception occurs and you need to abort the transaction, some statements will never actually be issued. Furthermore, this keeps lock times in the database as short as possible (from the late UPDATE to the transaction end).

- **Never updating unmodified objects.** It is very common in hand-coded JDBC to see the persistent state of an object updated, just in case it changed.....for example, the user pressed the save button but may not have edited any fields. Hibernate always knows if an object's state *actually* changed, as long as you are inside the same (possibly very long) unit of work.

- **Efficient Collection Handling.** Likewise, Hibernate only ever inserts/updates/deletes collection rows that actually changed.

- **Rolling two updates into one.** As a corollary to (1) and (3), Hibernate can roll two seemingly unrelated updates of the same object into one UPDATE statement.

- **Updating only the modified columns.** Hibernate knows exactly which columns need updating and, if you choose, will update only those columns.

- **Outer join fetching.** Hibernate implements a very efficient outer-join fetching algorithm! In addition, you can use *subselect* and *batch* pre-fetch optimizations.

- **Lazy collection initialization.**

- **Lazy object initialization.** Hibernate can use runtime-generated proxies (CGLIB) or interception injected through bytecode instrumentation at build-time.

A few more (optional) features of Hibernate that your handcoded JDBC may or may not currently benefit from

- second-level caching of arbitrary query results, from HQL, Criteria, and even native SQL queries
- efficient 'PreparedStatement' caching (Hibernate *always* uses 'PreparedStatement' for calls to the database)
- JDBC 2 style batch updates
- Pluggable connection pooling

Why Hibernate?

- **Natural Programming Model:** Hibernate lets you develop persistent classes following natural Object-oriented idioms including inheritance, polymorphism, association, composition, and the Java collections framework.

- **Transparent Persistence:** Hibernate requires no interfaces or base classes for persistent classes and enables any class or data structure to be persistent. Furthermore, Hibernate enables faster build procedures since it does not introduce build-time source or byte code generation or processing.

- **High Performance:** Hibernate supports lazy initialization, many fetching strategies, and optimistic locking with automatic versioning and time stamping. Hibernate requires no special database tables or fields and generates much of the SQL at system initialization time instead of runtime. Hibernate consistently offers superior performance over straight JDBC coding.

- **Reliability and Scalability:** Hibernate is well known for its excellent stability and quality, proven by the acceptance and use by tens of thousands of Java developers. Hibernate was designed to work in an application server cluster and deliver a highly scalable architecture. Hibernate scales well in any environment: Use it to drive your in-house Intranet that serves hundreds of users or for mission-critical applications that serve hundreds of thousands.

- **Extensibility:** Hibernate is highly customizable and extensible.

- **Comprehensive Query Facilities:** Including support for Hibernate Query Language (HQL), Java Persistence Query Language (JPAQL), Criteria queries, and "native SQL" queries; all of which can be scrolled and paginated to suit your exact performance needs.

Advantages of Hibernate

- **Caching objects.** The session is a transaction-level cache of persistent objects. You may also enable a JVM-level/cluster cache to memory and/or local disk.

- **Executing SQL statements later,** when needed. The session never issues an INSERT or UPDATE until it is actually needed. So if an exception occurs and you need to abort the transaction, some statements will never actually be issued. Furthermore, this keeps lock times in the database as short as possible (from the late UPDATE to the transaction end).

- **Never updating unmodified objects.** It is very common in hand-coded JDBC to see the persistent state of an object updated, just in case it changed.....for example, the user pressed the save button but may not have edited any fields. Hibernate always knows if an object's state *actually* changed, as long as you are inside the same (possibly very long) unit of work.

- **Efficient Collection Handling.** Likewise, Hibernate only ever inserts/updates/deletes collection rows that actually changed.

- **Rolling two updates into one.** As a corollary to (1) and (3), Hibernate can roll two seemingly unrelated updates of the same object into one UPDATE statement.

- **Updating only the modified columns.** Hibernate knows exactly which columns need updating and, if you choose, will update only those columns.

- **Outer join fetching.** Hibernate implements a very efficient outer-join fetching algorithm! In addition, you can use *subselect* and *batch* pre-fetch optimizations.

- **Lazy collection initialization.**
- **Lazy object initialization.** Hibernate can use runtime-generated proxies (CGLIB) or interception injected through bytecode instrumentation at build-time.

A few more (optional) features of Hibernate that your handcoded JDBC may or may not currently benefit from

- second-level caching of arbitrary query results, from HQL, Criteria, and even native SQL queries
- efficient 'PreparedStatement' caching (Hibernate *always* uses 'PreparedStatement' for calls to the database)
- JDBC 2 style batch updates
- Pluggable connection pooling

<u>Note:</u> How to acheive concurrency in Hibernate ?

Consider a scenario in which multiple applications are accessing same database. It is likely that is a data or row is fetched by an application1 and later on by application2 and application2 makes an update in DB and when application1 makes update then if finds that the old data is lost and not sure what to do.

To maintain high concurrency and high scalability hibernate allows the concept of versioning.Version checking uses version numbers, or timestamps, to detect conflicting updates (and to prevent lost updates). Hibernate provides for three possible approaches to writing application code that uses optimistic concurrency.

1) **Application version checking**: each interaction with the database occurs in a new Session and the developer is responsible for reloading all persistent instances from the database before manipulating them. This approach forces the application to carry out its own version checking to ensure conversation transaction isolation.
This approach is the least efficient in terms of database access.

```
// foo is an instance loaded by a previous Session
session = factory.openSession();
Transaction t = session.beginTransaction();
int oldVersion = foo.getVersion();
session.load( foo, foo.getKey() ); // load the current state
if ( oldVersion != foo.getVersion() ) throw new StaleObjectStateException();
foo.setProperty("bar");
t.commit();
session.close();
```

The version property is mapped using <version>, and Hibernate will automatically increment it during flush if the entity is dirty.
Clearly, manual version checking is only feasible in very trivial circumstances and not practical for most applications.

2) **Extended session and automatic versioning** : A single Session instance and its persistent instances are used for the whole conversation, known as session-per-conversation. Hibernate checks instance versions at flush time, throwing an exception if concurrent modification is detected. It’s up to the developer to catch and handle this exception.

The Session is disconnected from any underlying JDBC connection when waiting for user interaction. This approach is the most efficient in terms of database access. The application need not concern itself with version checking or with reattaching detached instances, nor does it have to reload instances in every database transaction.

```
// foo is an instance loaded earlier by the old session
Transaction t = session.beginTransaction(); // Obtain a new JDBC connection, start
transaction
foo.setProperty("bar");
session.flush(); // Only for last transaction in conversation
t.commit(); // Also return JDBC connection
session.close(); // Only for last transaction in conversation
```

3) **Detached objects and automatic versioning:** 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 reattaches them using Session.update(), Session.saveOrUpdate(), or Session.merge().

```
// foo is an instance loaded by a previous Session
foo.setProperty("bar");
```

```
session = factory.openSession();
Transaction t = session.beginTransaction();
session.saveOrUpdate(foo); // Use merge() if "foo" might have been loaded already
t.commit();
session.close();
```

Again, Hibernate will check instance versions during flush, throwing an exception if conflicting updates occurred.

4) **Customizing automatic versioning**
You may disable Hibernate's automatic version increment for particular properties and collections by setting the optimistic-lock mapping attribute to false. Hibernate will then no longer increment versions if the property is dirty.

Legacy database schemas are often static and can't be modified. Or, other applications might also access the same database and don't know how to handle version numbers or even timestamps. In both cases, versioning can't rely on a particular column in a table. To force a version check without a version or timestamp property mapping, with a comparison of the state of all fields in a row, turn on optimistic-lock="all" in the <class> mapping Sometimes concurrent modification can be permitted as long as the changes that have been made don't overlap. If you set optimistic-lock="dirty" when mapping the <class>, Hibernate will only compare dirty fields during flush .

List of Cache Providers in Hibernate and their Concurrency Support details

C**ache providers.**

Cache	**Provider class**	**Type**	**Cluster Safe**	**Query Cache Supported**
Hashtable (not intended for production use)	org.hibernate.cache. HashtableCache Provider	memory		yes
EHCache	org.hibernate.cache. EhCacheProvider	memory, disk		yes
OSCache	org.hibernate.cache. OSCacheProvider	memory, disk		yes
SwarmCache	org.hibernate.cache. SwarmCacheProvider	clustered (ip multicast)	yes (clustered invalidation)	yes (clock sync req.)
JBoss Cache 1.x	org.hibernate.cache. TreeCacheProvider	clustered (ip multicast), transactional	yes (replication)	yes (clock sync req.)
JBoss Cache 2	org.hibernate.cache. jbc2.JBossCache RegionFactory	clustered (ip multicast), transactional	yes (replication or invalidation)	yes (clock sync req.)

Cache Concurrency Strategy Support

Cache	**read-only**	**nonstrict-read-write**	**read-write**	**transactional**
Hashtable (not intended for production use)	yes	yes	yes	
EHCache	yes	yes	yes	
OSCache	yes	yes	yes	
SwarmCache	yes	yes		
JBoss Cache 1.x	yes			yes
JBoss Cache 2	yes			yes

EHCache (Easy Hibernate Cache)

(org.hibernate.cache.EhCacheProvider)

- It is **fast**.
- **lightweight**.
- **Easy-to-use**.
- Supports **read-only** and **read/write** caching.
- Supports **memory-based** and **disk-based** caching.
- Does not support **clustering**.

OSCache (Open Symphony Cache)

(org.hibernate.cache.OSCacheProvider)

- It is a **powerful** .
- **flexible** package
- supports **read-only** and **read/write** caching.
- Supports **memory- based** and **disk-based** caching.
- Provides basic support for **clustering** via either **JavaGroups or JMS.**

SwarmCache (org.hibernate.cache.SwarmCacheProvider)

- is a **cluster-based** caching.
- supports **read-only** or **nonstrict read/write** caching .
- appropriate for applications those have **more read** operations than **write** operations.

JBoss TreeCache (org.hibernate.cache.TreeCacheProvider)

- is a **powerful replicated** and **transactional** cache.
- useful when we need a true **transaction-capable** caching architecture .

The Second Level Cache in Hibernate : Settings and Configurations.

A Hibernate Session is a transaction-level cache of persistent data. We can configure a cluster or JVM-level (SessionFactory-level) cache on a class-by-class and collection-by-collection basis. We can also plug in a clustered cache into Hibernate. At the time of providing cache we need to understand that when we are updating the persistence DB it will not automatically reflect on Cache.

Configuring CACHE in Hibernate

We need to tell Hibernate that which caching implementation we need to use. This we can accomplish by specifying the name of a class that implements **org.hibernate.cache.CacheProvider** using the property **hibernate.cache.provider_class**. Hibernate comes bundled with a number of built-in integrations with open-source cache providers; additionally, we could implement your own and plug it in as outlined above. Prior to 3.2 Hibernate is defaulted to use EhCache as the default cache provider.

To find the CACHE PROVIDERS

Cache mappings

The <cache> element of a class or collection mapping has the following form:

```
<cache
    usage="transactional|read-write|nonstrict-read-write|read-only"  (1)
    region="RegionName"                                              (2)
    include="all|non-lazy"                                           (3)
/>
```

(1)

`usage` (required) specifies the caching strategy: `transactional`, `read-write`, `nonstrict-read-write` or `read-only`

(2)

`region` (optional, defaults to the class or collection role name) specifies the name of the second level cache region

(3)

`include` (optional, defaults to `all`) `non-lazy` specifies that properties of the entity mapped with `lazy="true"` may not be cached when attribute-level lazy fetching is enabled

Strategy: read only

- Useful for data that is **read frequently but never updated**.
- It is **Simple** .
- Best performer among the all.

If application needs to read but never modify instances of a persistent class, a `read-only` cache may be used. This is the simplest and best performing strategy. It's even perfectly safe for use in a cluster.

```
<class name="eg.Immutable" mutable="false">
    <cache usage="read-only"/>
    ....
</class>
```

Strategy: read/write

- Used when our **data** needs to be **updated**.
- It's having more overhead than **read-only** caches.
- When **Session.close()** or **Session.disconnect()** is called the transaction should be completed in an environment where JTA is no used.
- It is never used if **serializable transaction isolation level** is required.
- In a JTA environment, for obtaining the JTA **TransactionManager** we must specify the property**hibernate.transaction.manager_lookup_class**.
- To use it in a **cluster** the cache implementation must support **locking**.

If the application needs to update data, a `read-write` cache might be appropriate. This cache strategy should never be used if serializable transaction isolation level is required. If the cache is used in a JTA environment, you must specify the property `hibernate.transaction.manager_lookup_class`, naming a strategy for obtaining the JTA `TransactionManager`. In other environments, you should ensure that the transaction is completed when `Session.close()` or `Session.disconnect()` is called. If you wish to use this strategy in a cluster, you should ensure that the underlying cache implementation supports locking. The built-in cache providers do *not*.

```
<class name="eg.Cat" .... >
    <cache usage="read-write"/>
    ....
    <set name="kittens" ... >
        <cache usage="read-write"/>
        ....
    </set>
</class>
```

Strategy: nonstrict read/write

- Needed if the application needs to **update data rarely**.
- we must specify **hibernate.transaction.manager_lookup_class** to use this in a JTA environment .
- The transaction is completed when **Session.close()** or **Session.disconnect()** is called In other environments (except JTA) .

If the application only occasionally needs to update data (ie. if it is extremely unlikely that two transactions would try to update the same item simultaneously) and strict transaction isolation is not required, a `nonstrict-read-write` cache might be appropriate. If the cache is used in a JTA environment, you must specify `hibernate.transaction.manager_lookup_class`. In other environments, you should ensure that the transaction is completed when `Session.close()` or `Session.disconnect()` is called.

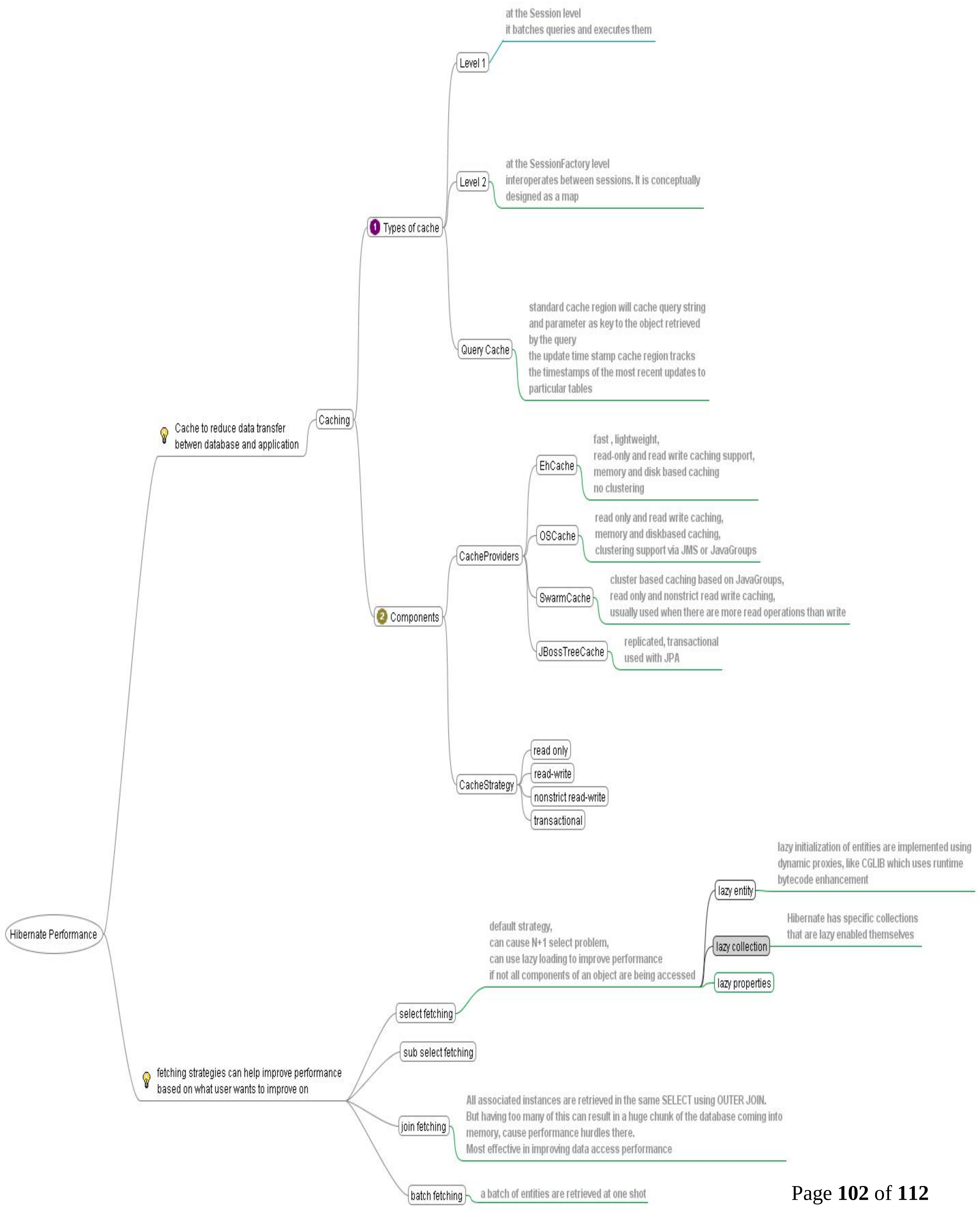
Strategy: transactional

- It supports only transactional cache providers such as **JBoss TreeCache**.
- only used in **JTA environment**.

The `transactional` cache strategy provides support for fully transactional cache providers such as JBoss TreeCache. Such a cache may only be used in a JTA environment and you must specify `hibernate.transaction.manager_lookup_class`.

Improving performance in hibernate

Hibernate has many performance improvement techniques, of course we have implemented a small sub set of that for our task. Let me first talk about Hibernate’s performance improvement strategies. If you want to take it all in at a glance, take a look at this mind map image.


Hibernate Performance
Cache to reduce data transfer betwen database and application
Caching
1 Types of cache
Level 1
at the Session level
it batches queries and executes them
Level 2
at the SessionFactory level
interoperates between sessions. It is conceptually designed as a map
Query Cache
standard cache region will cache query string and parameter as key to the object retrieved by the query
the update time stamp cache region tracks the timestamps of the most recent updates to particular tables
2 Components
CacheProviders
EhCache
fast , lightweight,
read-only and read write caching support,
memory and disk based caching
no clustering
OSCache
read only and read write caching,
memory and diskbased caching,
clustering support via JMS or JavaGroups
SwarmCache
cluster based caching based on JavaGroups,
read only and nonstrict read write caching,
usually used when there are more read operations than write
JBossTreeCache
replicated, transactional
used with JPA
CacheStrategy
read only
read-write
nonstrict read-write
transactional
fetching strategies can help improve performance based on what user wants to improve on
select fetching
default strategy,
can cause N+1 select problem,
can use lazy loading to improve performance
if not all components of an object are being accessed
lazy entity
lazy initialization of entities are implemented using dynamic proxies, like CGLIB which uses runtime bytecode enhancement
lazy collection
Hibernate has specific collections that are lazy enabled themselves
lazy properties
sub select fetching
join fetching
All associated instances are retrieved in the same SELECT using OUTER JOIN.
But having too many of this can result in a huge chunk of the database coming into memory, cause performance hurdles there.
Most effective in improving data access performance
batch fetching
a batch of entities are retrieved at one shot

First we need to understand that

- Sessionfactory is an immutable thread safe factory that initalize JDBC connections, connection pools and create Sessions.
- Session is a non thread safe single unit of work that represents a transaction

Caching, a blessing in disguise
Caching reduces traffic between the database and application by conserving data that has already been loaded into the application. Caches store data that was already fetched so that multiple accesses on the same data takes lesser time. Essentially caching reduces disk access, reduces computation time and speeds up response to users.

<u>Hibernate uses three levels of caching.</u>

- Level 1 mainly caches at the Session level
- Level 2 cache does it as the SessionFactory level.
- Query cache

Hibernate uses Level 1 cache to mainly reduce the number of SQL queries. It is always the default cache. If there are several modification on the same object it will simply generate a single SQL query for this. The level 1 cache is usually restrained to be for a single session, it is short lived. Essentially the general idea behind the fist level cache is that it batches queries.

A Level 2 cache is designed to interoperate between sessions. Level 2 cache is usually recommended when we are dealing with read only objects. It is not enabled by default. It is conceptually a map that has the id of the object as the key and the set of attributes the entity has as the value.

<u>The Query cache is not on by default either. It uses two cache regions –</u>

- StandardQueryCache - stores the query along with the parameters as key to the cache region. So any subsequent queries with the same key will hit the query cache and retrieve the object from the cache.

- UpdateTimeStampsCache - tracks the timestamps of the most recent updates to particular tables to identify stale results.

Remember all this caching will only be effective in reducing the number of queries if we use session.get to load the object. Using HQL to load the object may in fact create more queries.

<u>Hibernate has four basic types of cache providers-</u>

- EHCache - fast, lightweight, read-only and read write caching support,memory and disk based caching , no clustering.

- OSCache - read only and read write caching, memory and disk based caching, clustering support via JMS or JavaGroups.

- SwarmCache - cluster based caching based on JavaGroups, read only and nonstrict read write caching, usually used when there are more read operations than write.

- JBoss TreeCache - replicated and transactional cache.

- Tangosol Coherence Cache

The caching strategy is specified using a <cache usage = “”> tag. The caching strategies maybe:

- read only - for frequently read data, simple, best performer.

- read-write - data needs to be updated, never used if serializable transaction isolation level is needed, need to specify a manager_lookup_class in JTA environment.
- nonstrict read-write - rarely updating data , need to specify a manager_lookup_class in JTA environment.
- transactional - only used in a JTA environment

If the hibernate.cache.provider_class property is set, second level cache is enabled. Cache can be configured within hibernate.cfg.xml. Cache's usage patterns can be defined within the <cache> element in the hbm's associated with each domain class. Enable query caching by setting hibernate.cache.use_query_cache to true and call the setCacheable(true) on the Query object. Query cache always uses the second level cache. The Cache is loaded whenever an object is passed to save(), update(), saveOrUpdate() or when retrieving objects using load(), get(), list(). Invoking flush() will synchronize the object with the database. Use evict() to remove it from cache. A CacheMode defines how a particular session interacts with second level cache -

NORMAL - read and write to cache,
GET - read but dont put,
PUT - write but dont read,
REFRESH - force refresh of cache for all items read fromt he database

Fetching strategies
A fetching strategy identifies how hibernate will fetch an object along with it's associations once a query is executed. There are four types of strategies

- Join Fetching - All associated instances are retrieved in the same SELECT using OUTER JOIN. But having too many of this can result in a huge chunk of the database coming into memory, cause performance hurdles there.

- Select Fetching - This is the default strategy. A second SELECT retrieves associated entity or collection. This is usually lazy unless specified otherwise. This is extremely vulnerable to the N+1 select problem, so instead the join fetching can be enabled.

- Subselect fetching - similar to select but retrieves associated collections for all entries fetched previously.

- Batch fetching - optimization on select fetching where a batch of entities are retrieved in one select.

As far as we are concerned, we pretty much use EHCache as our caching strategy and do a lot of join fetching / lazy select fetching based on our requirements.

Hibernate Caches

Background

Hibernate comes with three different caching mechanisms - first level, second level and query cache. Truly understanding how the Hibernate caches work and interact with each other is important when you need to increase performance - just enabling caching in your entity with an annotation (or in classic .hbm.xml mapping file) is easy. But understanding what and how things happens behind the scenes is not. You might even end up with a less performing system if you do not know what you are doing.

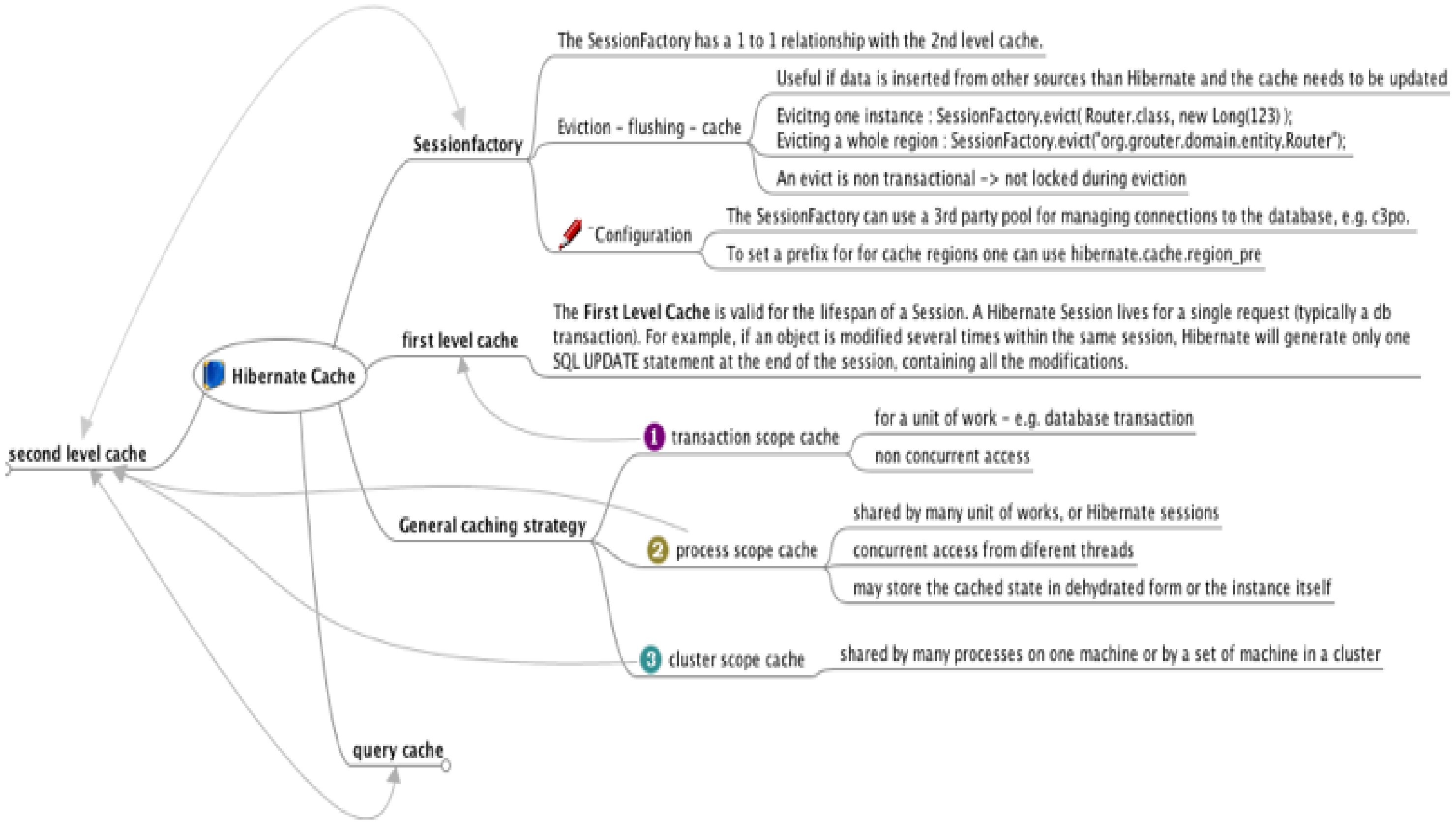
SessionFactory and Session

The purpose of the Hibernate SessionFactory (called EntityManager in JEE) is to create Sessions, initialize JDBC connections and pool them (using a pluggable provider like C3P0). A SessionFactory is immutable and built from a Configuration holding mapping information, cache information and a lot of other information usually provided by means of a hibernate.cfg.cml file or through a Spring bean configuration.

A Session is a unit of work at its lowest level - representing a transaction in database lingua. When a Session is created and operations are done on Hibernate entities, e.g. setting an attribute of an entity, Hibernate does not go of and update the underlying table immediately. Instead Hibernate keeps track of the state of an entity, whether it is dirty or not, and flushes (commits) updates at the end at the end of a unit of work. This is what Hibernate calls the first level cache.

The 1st level cache

Definition: The first level cache is where Hibernate keeps track of the possible dirty states of the ongoing Session's loaded and touched entities. The ongoing Session represents a unit of work and is always used and can not be turned of. The purpose of the first level cache is to hinder to many SQL queries or updates being made to the database, and instead batch them together at the end of the Session. When you think about the 1st level cache think Session.


Hibernate Cache
Sessionfactory
The SessionFactory has a 1 to 1 relationship with the 2nd level cache.
Eviction - flushing - cache
Useful if data is inserted from other sources than Hibernate and the cache needs to be updated
Evicitng one instance : SessionFactory.evict(Router.class, new Long(123));
Evicting a whole region : SessionFactory.evict("org.grouter.domain.entity.Router");
An evict is non transactional -> not locked during eviction
Configuration
The SessionFactory can use a 3rd party pool for managing connections to the database, e.g. c3po.
To set a prefix for for cache regions one can use hibernate.cache.region_pre
first level cache
The First Level Cache is valid for the lifespan of a Session. A Hibernate Session lives for a single request (typically a db transaction). For example, if an object is modified several times within the same session, Hibernate will generate only one SQL UPDATE statement at the end of the session, containing all the modifications.
second level cache
General caching strategy
1 transaction scope cache
for a unit of work - e.g. database transaction
non concurrent access
2 process scope cache
shared by many unit of works, or Hibernate sessions
concurrent access from diferent threads
may store the cached state in dehydrated form or the instance itself
3 cluster scope cache
shared by many processes on one machine or by a set of machine in a cluster
query cache

The 2nd level cache
The 2nd level cache is a process scoped cache that is associated with one SessionFactory. It will survive Sessions and can be reused in new Session by same SessionFactory (which usually is one per application). By default the 2nd level cache is not enabled.

The hibernate cache does not store instances of an entity - instead Hibernate uses something called dehydrated state. A dehydrated state can be thought of as a deserialized entity where the dehydrated state is like an array of strings, integers etc and the id of the entity is the pointer to the dehydrated entity. Conceptually you can think of it as a Map which contains the id as key and an array as value. Or something like below for a cache region:

```
{ id -> { atribute1, attribute2, attribute3 } }
{ 1 -> { "a name", 20, null } }
{ 2 -> { "another name", 30, 4 } }
```

If the entity holds a collection of other entities then the other entity also needs to be cached. In this case it could look something like:

```
{ id -> { atribute1, attribute2, attribute3, Set{item1..n} } }
{ 1 -> { "a name", 20, null , {1,2,5} } }
{ 2 -> { "another name", 30, 4 {4,8}} }
```

The actual implementation of the 2nd level cache is not done by Hibernate (there is a simple Hashtable cache available, not aimed for production though). Hibernate instead has a plugin concept for caching providers which is used by e.g. EHCache.

Enabling the 2nd level cache and EHCache

To get the 2nd level cache working you need to do 2 things:
1 Cache Strategy. Enable a cache strategy for your Hibernate entity - either in the class with an annotation or in the hibernate mapping xml file if you are stuck with pre java5. This can be done for an entity by providing this little snippet into your hbm.xml file (a better place is to store the cache setting strategy in hibernate.cg.xml file)

```
<class name="org.grouter.domain.entities.Router" table="ROUTER">
<cache usage="transactional|read-write|nonstrict-read-write|read-only" />
<id ...
</class>
```

or using an annotation for your entity (if you are on java5 or greater)

```
@Entity
@Cache(usage = CacheConcurrencyStrategy.NONSTRICT_READ_WRITE)
public class Router { ... }
```

And as mentioned above if you want to cache collections of an entity you need to specify caching on collection level:

```
<class name="org.grouter.domain.entities.Router" table="ROUTER">
    <cache usage="transactional|read-write|nonstrict-read-write|read-only"/>
    <id ...
    <set name="nodes">
        <cache usage="transactional|read-write|nonstrict-read-write|read-only"/>
        ...
    </set>
</class>
```

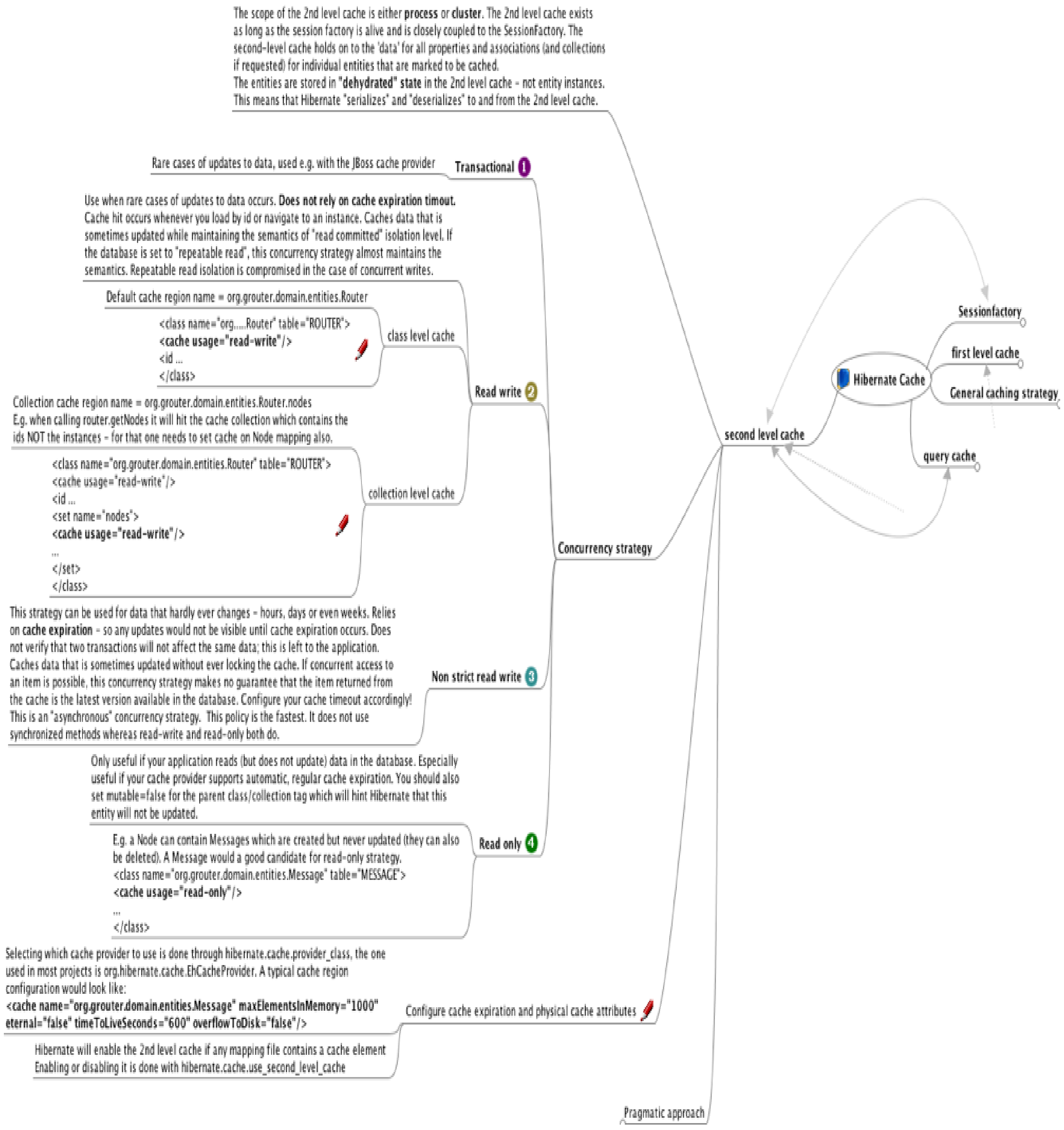
Hibernate has something called a cache region which by default will be the full qualified name of your Java class. And if you like me are a fan of convention over configuration you will use the default region for an entity. A cache region will also be needed for the collection using the full qualified name of the Java class plus the name of the collection name (i.e. org.grouter.domain.entities.Router.nodes)

2 Cache provider. Setting up the physical caching for a cache provider. If you are using EHCache - which is the most common choice i dear to say - then you will need to specify some settings for the cache regions of your entities in a file called ehcache.xml. The EHCache will look for this file in the classpath and if not found it will fallback to ehcache-failsafe.xml which resides in the ehcache.jar library A typical sample for an EHCache configuration could look like (see mind map below for explanations):

```
<cache name="org.grouter.domain.entities.Router" maxElementsInMemory="1000" eternal="false"
timeToLiveSeconds="600" overflowToDisk="false"/>
and
<cache name="org.grouter.domain.entities.Router.nodes" maxElementsInMemory="1000" eternal="false"
timeToLiveSeconds="600" overflowToDisk="false"/>
```

The name maps to the name of the cache region of your entity. The attribute maxelementsInMemory needs to be set so that Hibernate does not have to swap in and out elements from the cache. A good choice for a read only cache would be as many entities there are in the database table the entity represents. The attribute eternal, if set to true means that any time outs specified will be ignored and entities put into the cache from Hibernate will live for ever.

Below is a mindmap for the second level cache and how it relates to the SessionFactory and the 1st level cache.


The scope of the 2nd level cache is either **process** or **cluster**. The 2nd level cache exists as long as the session factory is alive and is closely coupled to the SessionFactory. The second-level cache holds on to the 'data' for all properties and associations (and collections if requested) for individual entities that are marked to be cached.
The entities are stored in **"dehydrated" state** in the 2nd level cache - not entity instances. This means that Hibernate "serializes" and "deserializes" to and from the 2nd level cache.

Transactional 1

Rare cases of updates to data, used e.g. with the JBoss cache provider

Read write 2

Use when rare cases of updates to data occurs. **Does not rely on cache expiration timout.** Cache hit occurs whenever you load by id or navigate to an instance. Caches data that is sometimes updated while maintaining the semantics of "read committed" isolation level. If the database is set to "repeatable read", this concurrency strategy almost maintains the semantics. Repeatable read isolation is compromised in the case of concurrent writes.

class level cache

Default cache region name = org.grouter.domain.entities.Router

```
<class name="org.....Router" table="ROUTER">
<cache usage="read-write"/>
<id ...
</class>
```

collection level cache

Collection cache region name = org.grouter.domain.entities.Router.nodes
E.g. when calling router.getNodes it will hit the cache collection which contains the ids NOT the instances - for that one needs to set cache on Node mapping also.

```
<class name="org.grouter.domain.entities.Router" table="ROUTER">
<cache usage="read-write"/>
<id ...
<set name="nodes">
<cache usage="read-write"/>
...
</set>
</class>
```

Non strict read write 3

This strategy can be used for data that hardly ever changes - hours, days or even weeks. Relies on **cache expiration** - so any updates would not be visible until cache expiration occurs. Does not verify that two transactions will not affect the same data; this is left to the application. Caches data that is sometimes updated without ever locking the cache. If concurrent access to an item is possible, this concurrency strategy makes no guarantee that the item returned from the cache is the latest version available in the database. Configure your cache timeout accordingly! This is an "asynchronous" concurrency strategy. This policy is the fastest. It does not use synchronized methods whereas read-write and read-only both do.

Read only 4

Only useful if your application reads (but does not update) data in the database. Especially useful if your cache provider supports automatic, regular cache expiration. You should also set mutable=false for the parent class/collection tag which will hint Hibernate that this entity will not be updated.

E.g. a Node can contain Messages which are created but never updated (they can also be deleted). A Message would a good candidate for read-only strategy.

```
<class name="org.grouter.domain.entities.Message" table="MESSAGE">
<cache usage="read-only"/>
...
</class>
```

Concurrency strategy

Configure cache expiration and physical cache attributes

Selecting which cache provider to use is done through hibernate.cache.provider_class, the one used in most projects is org.hibernate.cache.EhCacheProvider. A typical cache region configuration would look like:

```
<cache name="org.grouter.domain.entities.Message" maxElementsInMemory="1000"
eternal="false" timeToLiveSeconds="600" overflowToDisk="false"/>
```

Hibernate will enable the 2nd level cache if any mapping file contains a cache element
Enabling or disabling it is done with hibernate.cache.use_second_level_cache

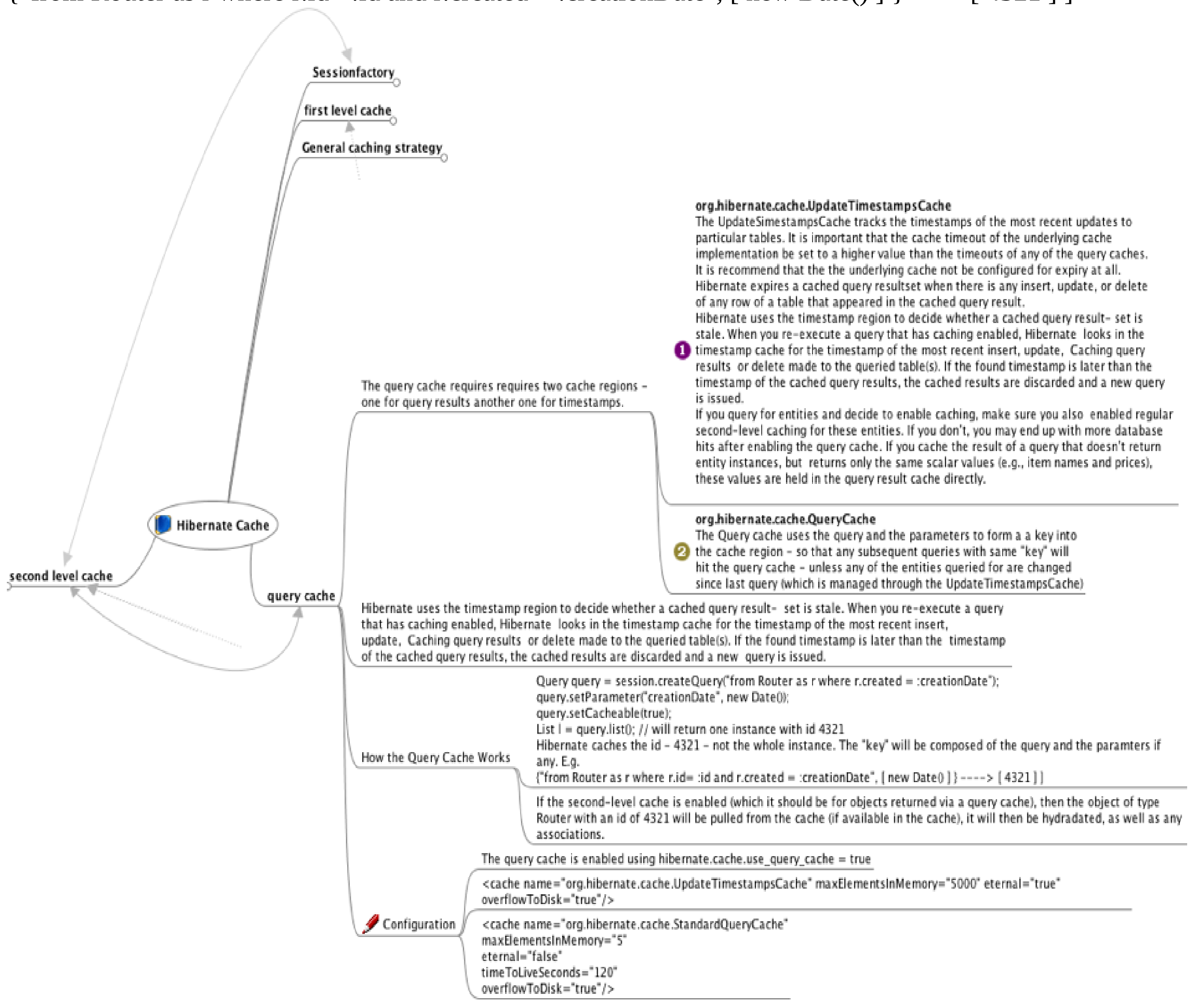
Pragmatic approach

second level cache

The Query cache
The Query cache of Hibernate is not on by default. It uses two cache regions called org.hibernate.cache.StandardQueryCache and org.hibernate.cache.UpdateTimestampsCache. The first one stores the query along with the parameters to the query as a key into the cache and the last one keeps track of stale query results. If an entity part of a cached query is updated the the query cache evicts the query and its cached result from the query cache. Of course to utilize the Query cache the returned and used entities must be set using a cache strategy as discussed previously. A simple load(id) will not use the query cache but instead if you have a query like:

```
Query query = session.createQuery("from Router as r where r.created = :creationDate");
query.setParameter("creationDate", new Date());
query.setCacheable(true);
List l = query.list(); // will return one instance with id 4321
```

Hibernate will cache using as key the query and the parameters the value of the if of the entity.
{ query,{parameters}} ---> {id of cached entity}
{"from Router as r where r.id= :id and r.created = :creationDate", [new Date()] } ----> [4321]]



The query cache requires requires two cache regions - one for query results another one for timestamps.

org.hibernate.cache.UpdateTimestampsCache
The UpdateSimestampsCache tracks the timestamps of the most recent updates to particular tables. It is important that the cache timeout of the underlying cache implementation be set to a higher value than the timeouts of any of the query caches. It is recommend that the the underlying cache not be configured for expiry at all. Hibernate expires a cached query resultset when there is any insert, update, or delete of any row of a table that appeared in the cached query result.
Hibernate uses the timestamp region to decide whether a cached query result- set is stale. When you re-execute a query that has caching enabled, Hibernate looks in the timestamp cache for the timestamp of the most recent insert, update, Caching query results or delete made to the queried table(s). If the found timestamp is later than the timestamp of the cached query results, the cached results are discarded and a new query is issued.
If you query for entities and decide to enable caching, make sure you also enabled regular second-level caching for these entities. If you don't, you may end up with more database hits after enabling the query cache. If you cache the result of a query that doesn't return entity instances, but returns only the same scalar values (e.g., item names and prices), these values are held in the query result cache directly.

org.hibernate.cache.QueryCache
The Query cache uses the query and the parameters to form a a key into the cache region - so that any subsequent queries with same "key" will hit the query cache - unless any of the entities queried for are changed since last query (which is managed through the UpdateTimestampsCache)

Hibernate uses the timestamp region to decide whether a cached query result- set is stale. When you re-execute a query that has caching enabled, Hibernate looks in the timestamp cache for the timestamp of the most recent insert, update, Caching query results or delete made to the queried table(s). If the found timestamp is later than the timestamp of the cached query results, the cached results are discarded and a new query is issued.

```
Query query = session.createQuery("from Router as r where r.created = :creationDate");
query.setParameter("creationDate", new Date());
query.setCacheable(true);
List l = query.list(); // will return one instance with id 4321
```
Hibernate caches the id - 4321 - not the whole instance. The "key" will be composed of the query and the paramters if any. E.g.
{"from Router as r where r.id= :id and r.created = :creationDate", [new Date()] } ----> [4321]]

If the second-level cache is enabled (which it should be for objects returned via a query cache), then the object of type Router with an id of 4321 will be pulled from the cache (if available in the cache), it will then be hydradated, as well as any associations.

The query cache is enabled using hibernate.cache.use_query_cache = true

```
<cache name="org.hibernate.cache.UpdateTimestampsCache" maxElementsInMemory="5000" eternal="true"
overflowToDisk="true"/>
```

```
<cache name="org.hibernate.cache.StandardQueryCache"
maxElementsInMemory="5"
eternal="false"
timeToLiveSeconds="120"
overflowToDisk="true"/>
```

Pragmatic approach to the 2nd level cache
How do you now if you are hitting the cache or not? One way is using Hibernates SessionFactory to get statistics for cache hits. In your SessionFactory configuration you can enable the cache statistics by:

```
<prop key="hibernate.show_sql">true</prop>
<prop key="hibernate.format_sql">true</prop>
<prop key="hibernate.use_sql_comments">true</prop>
<prop key="hibernate.cache.use_query_cache">true</prop>
<prop key="hibernate.cache.use_second_level_cache">true</prop>
<prop key="hibernate.generate_statistics">true</prop>
<prop key="hibernate.cache.use_structured_entries">true</prop>
```

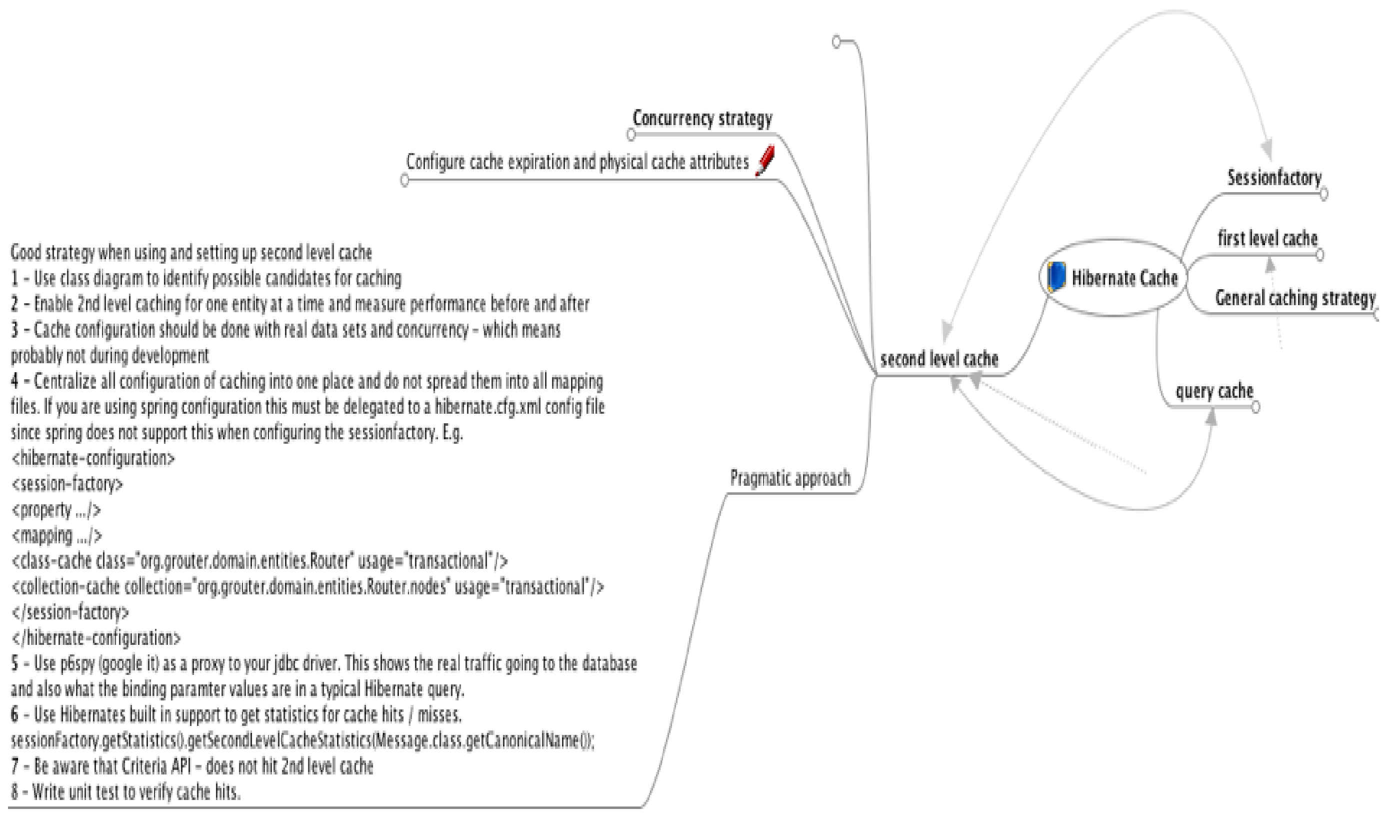
The you might want to write a unit test to verify that you indeed are hitting the cache. Below is some sample code where the unit test is extending Springs excellent AbstractTransactionalDataSourceSpringContextTests

```
public class MessageDAOTest extends AbstractDAOTests  // which extends
AbstractTransactionalDataSourceSpringContextTests
{
   public void testCache()
    {
        long numberOfMessages = jdbcTemplate.queryForInt("SELECT count(*) FROM message
");
        System.out.println("Number of rows :" + numberOfMessages);
        final String cacheRegion = Message.class.getCanonicalName();
        SecondLevelCacheStatistics settingsStatistics = sessionFactory.getStatistics().
            getSecondLevelCacheStatistics(cacheRegion);
        StopWatch stopWatch = new StopWatch();
        for (int i = 0; i < 10; i++)
        {
            stopWatch.start();
            messageDAO.findAllMessages();
            stopWatch.stop();
            System.out.println("Query time : " + stopWatch.getTime());
            assertEquals(0, settingsStatistics.getMissCount());
            assertEquals(numberOfMessages * i, settingsStatistics.getHitCount());
            stopWatch.reset();
            System.out.println(settingsStatistics);
            endTransaction();

            // spring creates a transaction when test starts - so we first end it then
start a new in the loop
            startNewTransaction();
        }
    }
}
```

The output could look something like:

```
30 Jan 08 23:37:14  INFO org.springframework.test.AbstractTransactionalSpringContextTests:323
- Began transaction (1): transaction manager
[org.springframework.orm.hibernate3.HibernateTransactionManager@ced32d]; default rollback =
true
Number of rows :6
Query time : 562
SecondLevelCacheStatistics[hitCount=0,missCount=0,putCount=6,elementCountInMemory=6,elem
entCountOnDisk=0,sizeInMemory=8814]

30 Jan 08 23:37:15  INFO org.springframework.test.AbstractTransactionalSpringContextTests:290
- Rolled back transaction
after test execution

30 Jan 08 23:37:15  INFO org.springframework.test.AbstractTransactionalSpringContextTests:323
- Began transaction (2):
transaction manager
[org.springframework.orm.hibernate3.HibernateTransactionManager@ced32d]; default rollback =
true
Query time : 8
SecondLevelCacheStatistics[hitCount=6,missCount=0,putCount=6,elementCountInMemory=6,elem
entCountOnDisk=0,sizeInMemory=8814]

30 Jan 08 23:37:15  INFO org.springframework.test.AbstractTransactionalSpringContextTests:290
- Rolled back transaction
after test execution

30 Jan 08 23:37:15  INFO org.springframework.test.AbstractTransactionalSpringContextTests:323
- Began transaction (3):
transaction manager
[org.springframework.orm.hibernate3.HibernateTransactionManager@ced32d]; default rollback =
true
Query time : 11
```

Another way to spy on what Hibernate is doing is to proxy the jdbc driver used by a proxy driver. One excellent one I use is p6spy which will show you exactly what is issued over a JDBC connection to the actual backend database. For other tips have a look below in the mindmap.



Good strategy when using and setting up second level cache
1 - Use class diagram to identify possible candidates for caching
2 - Enable 2nd level caching for one entity at a time and measure performance before and after
3 - Cache configuration should be done with real data sets and concurrency - which means probably not during development
4 - Centralize all configuration of caching into one place and do not spread them into all mapping files. If you are using spring configuration this must be delegated to a hibernate.cfg.xml config file since spring does not support this when configuring the sessionfactory. E.g.

```
<hibernate-configuration>
<session-factory>
<property .../>
<mapping .../>
<class-cache class="org.grouter.domain.entities.Router" usage="transactional"/>
<collection-cache collection="org.grouter.domain.entities.Router.nodes" usage="transactional"/>
</session-factory>
</hibernate-configuration>
```

5 - Use p6spy (google it) as a proxy to your jdbc driver. This shows the real traffic going to the database and also what the binding paramter values are in a typical Hibernate query.
6 - Use Hibernates built in support to get statistics for cache hits / misses.
sessionFactory.getStatistics().getSecondLevelCacheStatistics(Message.class.getCanonicalName());
7 - Be aware that Criteria API - does not hit 2nd level cache
8 - Write unit test to verify cache hits.