**Hibernate:**

For Disabling the second level of cache we have to made following change to hibernate configuration file.

1. <!-- Disable the second-level cache -->
2. **<property** name="cache.provider\_class"**>**org.hibernate.cache.NoCacheProvider**</property>**

For Enabling the second level of cache we have to made following change to hibernate configuration file.

1. <!-- Enable the second-level cache -->
2. **<property** name="cache.use\_second\_level\_cache"**>**true**</property>**
3. **<property** name="cache.provider\_class"**>**org.hibernate.cache.EhCacheProvider**</property>**

**Table A:** HQL Clauses with their description, syntax, and examples.

|  |  |  |  |
| --- | --- | --- | --- |
| **Clause** | **Description** | **Syntax** | **Example** |
| **from** | The simplest form of an HQL query. Specifies the object whose instances are to be returned as the query result. Commonly used with the **select** clause. | **from** *object* [**as** *object\_alias*]\* object\_alias simply means another name given to refer to an object for convenience. | from UserDetails as user  Will return all instances of object *UserDetails.* |
| **select** | Specifies objects and properties to be returned in the query result set. Used in conjunction with the **from** clause. | **select** [*object.*]*property* | select user.userName from UserDetails as user  Will return all values of userName in all instances of *UserDetails.* |
| **where** | Specifies the conditions that should be satisfied by the instances returned as the query result. Used with **select** and/or **from** clause. | **where** *condition*  Here, condition is a combination of logical, relational operators i.e. =, >, AND, NOT etc. | from UserDetails as user where user.userId > 2 Will return all instances of user in UserDetails whose correspondinguser.userId values are greater than 2. |
| **order by** | Specifies the order (**asc**ending/**desc**ending) in which the properties of objects returned as query results should be listed. Used with the **select** and **from** clauses. | **order by** *object0.property0*[**asc**|**desc**][, *object1.property0*]...  By default, order is ascending unless specified otherwise. | from UserDetails  as user order by userId asc  Will return a list of all instances of user in ascending order of corresponding userId values. |
| **group by** | Specifies the grouping criteria using objects properties, by which the list of objects returned as a query result should be grouped together. Used with the **select** and/or **from** clause. | **group by** *object0.property0*[, *object1.property0*]... | select userId from UserDetails as user group by user.userId   Will return list of all userId instances from user grouped by corresponding values of user. |

# Attributes Override:

Here we have seen that an Entity Type Object USER has a Value Type Object(or Embeddable Object ) ADDRESS with corresponding fields name street, city, pin-code and state save to the database table USER\_TABLE with value type object's column name (CITY\_NAME, PIN\_CODE, STATE\_NAME, STREET\_NAME).

But here some problems, suppose this user object have two types of addresses as like Local Address and Permanent Address then how to manage the column names of these value type objects in the database table USER\_TABLE. 

**@Column(name="ADDRESS")  
    @Embeddeds  
    @AttributeOverrides({  
    @AttributeOverride(name="street", column=@Column(name="HOME\_STREET\_NAME")),  
    @AttributeOverride(name="city", column=@Column(name="HOME\_CITY\_NAME")),  
    @AttributeOverride(name="state", column=@Column(name="HOME\_STATE\_NAME")),  
    @AttributeOverride(name="pincode", column=@Column(name="HOME\_PIN\_CODE"))})**  
    private Address homeAddress;  
      
    **@Embedded**  
    private Address permanentAddress;

To overcome this problem we have to override the Attributes of the Value type objects.

# Value Types and Embedding Objects in Hibernate

We’ll use the **@Embeddable** annotations to embed a value type object into our Entity class.

"An object of value type has no database identity; it belongs to an entity instance and its persistent state is embedded in the table row of the owning entity. Value types don't have identifiers or identifier properties"

Now In short we look the following points... 

**Object of Entity Type** : has its own database identity

**Object of Value Type** : belongs to an entity, and its persistent state is embedded in the table row of the owning entity. Value types don't have identifiers or identifier properties.

**@Embeddable:**

**Target:** Defines a class whose instances are stored as an intrinsic part of an owning entity and share the identity of the entity. Each of the persistent properties or fields of the embedded object is mapped to the database table for the entity.

**unsaved-value** - This is the value used to determine if a class has been made persistent. If the value of the id attribute is null, then it means that this object has not been persisted.

<**generator**> element:

Used to create primary key for new record, there are some commonly used generators type given below...

**Increment**- used to generate primary keys of type long, short or int that are unique only.

**Sequence**  - used to generate primary keys for DB2, Oracle, SAP Database.

**Assigned**  - is used when application code generates the primary key.

**Native**       - selects identity, sequence or Hilo depending upon the capabilities of the underlying db.

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

**Uuid**  - Unique use ID of 128 bits generated from using algorithm and return type is String

**hilo**  - generated by the hi/lo Algorithms

**seqhilo** - generated by the hi/lo Algorithm according to sequence of database

**select** - select from database triggered value

**foreign** - associated with the other model objects

**What is Dialect?**

Dialect is configuration specify here so hibernate knows whats kind of language we are used and what type database we are used. we can say it is database dependent. It connects the database specific query language which we want to use.

# Difference between HQL and Criteria Query in Hibernate?

Let us see the main differences between HQL and Criteria Query

* HQL is to perform both select and non-select operations on the data,  but Criteria is only for selecting the data, we cannot perform non-select operations using criteria
* HQL is suitable for executing Static Queries, where as Criteria is suitable for executing Dynamic Queries
* HQL doesn’t support pagination concept, but we can achieve pagination with Criteria
* Criteria used to take more time to execute then HQL
* With Criteria we are safe with SQL Injection because of its dynamic query generation but in HQL as your queries are either fixed or parameterized, there is no safe from SQL Injection.

Hibernate provides three different ways to retrieve data from database. We have already discussed [HQL and native SQL queries](http://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/). Now we will discuss our third option i.e. Criteria. The Criteria Query API lets you build nested, structured query expressions in Java, providing a compile-time syntax checking that is not possible with a query language like HQL or SQL. The Criteria API also includes query by example (QBE) functionality. This lets you supply example objects that contain the properties you would like to retrieve instead of having to step-by-step spell out the components of the query. It also includes projection and aggregation methods, including count(). Let’s explore it’s different features in detail.

**Table of Contents**

## Basic Usage Example

The Criteria API allows you to build up a criteria query object programmatically; the org.hibernate.Criteria interface defines the available methods for one of these objects. The Hibernate Session interface contains several createCriteria() methods. Pass the persistent object’s class or its entity name to the createCriteria() method, and Hibernate will create a Criteria object that returns instances of the persistence object’s class when your application executes a criteria query.

The simplest example of a criteria query is one with no optional parameters or restrictions—the criteria query will simply return every object that corresponds to the class.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  List<Product> results = crit.list(); |

Moving on from this simple example, we will add constraints to our criteria queries so we can whittle down the result set.

## Using Restrictions with Criteria

The Criteria API makes it easy to use restrictions in your queries to selectively retrieve objects; for instance, your application could retrieve only products with a price over $30. You may add these restrictions to a Criteria object with the add() method. The add() method takes an org.hibernate.criterion.Criterion object that represents an individual restriction. You can have more than one restriction for a criteria query.

**i) Restrictions.eq() Example**

To retrieve objects that have a property value that “**equals**” your restriction, use the eq() method on Restrictions, as follows:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.eq("description","Mouse"));  List<Product> results = crit.list() |

Above query will search all products having description as “Mouse”.

**ii) Restrictions.ne() Example**

To retrieve objects that have a property value “not equal to” your restriction, use the ne() method on Restrictions, as follows:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.ne("description","Mouse"));  List<Product> results = crit.list() |

Above query will search all products having description anything but not “Mouse”.

You cannot use the not-equal restriction to retrieve records with a NULL value in the database for that property (in SQL, and therefore in Hibernate, NULL represents the absence of data, and so cannot be compared with data). If you need to retrieve objects with NULL properties, you will have to use the isNull() restriction.

**iii) Restrictions.like() and Restrictions.ilike() Example**

Instead of searching for exact matches, we can retrieve all objects that have a property matching part of a given pattern. To do this, we need to create an SQL LIKE clause, with either the like() or the ilike() method. The ilike() method is case-insensitive.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.like("name","Mou%",MatchMode.ANYWHERE));  List<Product> results = crit.list(); |

Above example uses an org.hibernate.criterion.MatchMode object to specify how to match the specified value to the stored data. The MatchMode object (a type-safe enumeration) has four different matches:

**ANYWHERE: Anyplace in the string  
END: The end of the string  
EXACT: An exact match  
START: The beginning of the string**

**iv) Restrictions.isNull() and Restrictions.isNotNull() Example**

The isNull() and isNotNull() restrictions allow you to do a search for objects that have (or do not have) null property values.

|  |
| --- |
| **Criteria crit = session.createCriteria(Product.class);**  **crit.add(Restrictions.isNull("name"));**  **List<Product> results = crit.list();** |

**v) Restrictions.gt(), Restrictions.ge(), Restrictions.lt() and Restrictions.le() Examples**

Several of the restrictions are useful for doing math comparisons. The greater-than comparison is gt(), the greater-than-or-equal-to comparison is ge(), the less-than comparison is lt(), and the less-than-or-equal-to comparison is le(). We can do a quick retrieval of all products with prices over $25 like this, relying on Java’s type promotions to handle the conversion to Double:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.gt("price", 25.0));  List<Product> results = crit.list(); |

**vi) Combining Two or More Criteria Examples**

Moving on, we can start to do more complicated queries with the Criteria API. For example, we can combine AND and OR restrictions in logical expressions. When we add more than one constraint to a criteria query, it is interpreted as an AND, like so:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.lt("price",10.0));  crit.add(Restrictions.ilike("description","mouse", MatchMode.ANYWHERE));  List<Product> results = crit.list(); |

If we want to have two restrictions that return objects that satisfy either or both of the restrictions, we need to use the or() method on the Restrictions class, as follows:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  Criterion priceLessThan = Restrictions.lt("price", 10.0);  Criterion mouse = Restrictions.ilike("description", "mouse", MatchMode.ANYWHERE);  LogicalExpression orExp = Restrictions.or(priceLessThan, mouse);  crit.add(orExp);  List results=crit.list(); |

The orExp logical expression that we have created here will be treated like any other criterion. We can therefore add another restriction to the criteria:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  Criterion price = Restrictions.gt("price",new Double(25.0));  Criterion name = Restrictions.like("name","Mou%");  LogicalExpression orExp = Restrictions.or(price,name);  crit.add(orExp);  crit.add(Restrictions.ilike("description","blocks%"));  List results = crit.list(); |

**vii) Using Disjunction Objects with Criteria**

If we wanted to create an OR expression with more than two different criteria (for example, “price > 25.0 OR name like Mou% OR description not like blocks%”), we would use an org.hibernate.criterion.Disjunction object to represent a disjunction.

You can obtain this object from the disjunction() factory method on the Restrictions class. The disjunction is more convenient than building a tree of OR expressions in code. To represent an AND expression with more than two criteria, you can use the conjunction() method, although you can easily just add those to the Criteria object. The conjunction can be more convenient than building a tree of AND expressions in code. Here is an example that uses the disjunction:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  Criterion priceLessThan = Restrictions.lt("price", 10.0);  Criterion mouse = Restrictions.ilike("description", "mouse", MatchMode.ANYWHERE);  Criterion browser = Restrictions.ilike("description", "browser", MatchMode.ANYWHERE);  Disjunction disjunction = Restrictions.disjunction();  disjunction.add(priceLessThan);  disjunction.add(mouse);  disjunction.add(browser);  crit.add(disjunction);  List results = crit.list(); |

**viii) Restrictions.sqlRestriction() Example**

sqlRestriction() restriction allows you to directly specify SQL in the Criteria API. It’s useful if you need to use SQL clauses that Hibernate does not support through the Criteria API.

Your application’s code does not need to know the name of the table your class uses. Use {alias} to signify the class’s table, as follows:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.sqlRestriction("{alias}.description like 'Mou%'"));  List<Product> results = crit.list(); |

## Paging Through the ResultSet

One common application pattern that criteria can address is pagination through the result set of a database query. There are two methods on the Criteria interface for paging, just as there are for Query: setFirstResult() and setMaxResults(). The setFirstResult() method takes an integer that represents the first row in your result set, starting with row 0. You can tell Hibernate to retrieve a fixed number of objects with the setMaxResults() method. Using both of these together, we can construct a paging component in our web or Swing application.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.setFirstResult(1);  crit.setMaxResults(20);  List<Product> results = crit.list(); |

As you can see, this makes paging through the result set easy. You can increase the first result you return (for example, from 1, to 21, to 41, etc.) to page through the result set.

## Obtaining a Unique Result

Sometimes you know you are going to return only zero or one object from a given query. This could be because you are calculating an aggregate or because your restrictions naturally lead to a unique result. If you want obtain a single Object reference instead of a List, the uniqueResult() method on the Criteria object returns an object or null. If there is more than one result, the uniqueResult() method throws a HibernateException.

The following short example demonstrates having a result set that would have included more than one result, except that it was limited with the setMaxResults() method:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  Criterion price = Restrictions.gt("price",new Double(25.0));  crit.setMaxResults(1);  Product product = (Product) crit.uniqueResult(); |

Again, please note that you need to make sure that your query returns only one or zero results if you use the uniqueResult() method. Otherwise, Hibernate will throw a NonUniqueResultException exception.

## Obtaining Distinct Results

If you would like to work with distinct results from a criteria query, Hibernate provides a result transformer for distinct entities, org.hibernate.transform.DistinctRootEntityResultTransformer, which ensures that no duplicates will be in your query’s result set. **Rather than using SELECT DISTINCT with SQL, the distinct result transformer compares each of your results using their default hashCode() methods, and only adds those results with unique hash codes to your result set**. This may or may not be the result you would expect from an otherwise equivalent SQL DISTINCT query, so **be careful with this**.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  Criterion price = Restrictions.gt("price",new Double(25.0));  crit.setResultTransformer( DistinctRootEntityResultTransformer.INSTANCE )  List<Product> results = crit.list(); |

An additional performance note: the comparison is done in Hibernate’s Java code, not at the database, so non-unique results will still be transported across the network.

## Sorting the Query’s Results

Sorting the query’s results works much the same way with criteria as it would with HQL or SQL. The Criteria API provides the org.hibernate.criterion.Order class to sort your result set in either ascending or descending order, according to one of your object’s properties.

This example demonstrates how you would use the Order class:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.add(Restrictions.gt("price",10.0));  crit.addOrder(Order.desc("price"));  List<Product> results = crit.list(); |

You may add more than one Order object to the Criteria object. Hibernate will pass them through to the underlying SQL query. Your results will be sorted by the first order, then any identical matches within the first sort will be sorted by the second order, and so on. Beneath the covers, **Hibernate passes this on to an SQL ORDER BY clause after substituting the proper database column name for the property**.

## Performing Associations (Joins)

The association works when going from **either one-to-many or from many-to-one**. First, we will demonstrate how to use one-to-many associations to obtain suppliers who sell products with a price over $25. Notice that we create a new Criteria object for the products property, add restrictions to the products’ criteria we just created, and then obtain the results from the supplier Criteria object:

|  |
| --- |
| Criteria crit = session.createCriteria(Supplier.class);  Criteria prdCrit = crit.createCriteria("products");  prdCrit.add(Restrictions.gt("price",25.0));  List results = crit.list(); |

Going the other way, we obtain all the products from the supplier MegaInc using many-to-one associations:

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  Criteria suppCrit = crit.createCriteria("supplier");  suppCrit.add(Restrictions.eq("name","Hardware Are We"));  List results = crit.list(); |

## Adding Projections and Aggregates

Instead of working with objects from the result set, you can treat the results from the result set as a set of rows and columns, also known as a projection of the data. This is similar to how you would use data from a SELECT query with JDBC.

To use projections, start by getting the org.hibernate.criterion.Projection object you need from the org.hibernate.criterion.Projections factory class. The Projections class is similar to the Restrictions class in that it provides several static factory methods for obtaining Projection instances. After you get a Projection object, add it to your Criteria object with the setProjection() method. When the Criteria object executes, the list contains object references that you can cast to the appropriate type.

**Example 1 : Single Aggregate ( Getting Row Count )**

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  crit.setProjection(Projections.rowCount());  List<Long> results = crit.list(); |

Other aggregate functions available through the Projections factory class include the following:

1. **avg(String propertyName)**: Gives the average of a property’s value
2. **count(String propertyName)**: Counts the number of times a property occurs
3. **countDistinct(String propertyName)**: Counts the number of unique values the property contains
4. **max(String propertyName)**: Calculates the maximum value of the property values
5. **min(String propertyName)**: Calculates the minimum value of the property values
6. **sum(String propertyName)**: Calculates the sum total of the property values

**Example 2 : Multiple Aggregates**

We can apply more than one projection to a given Criteria object. To add multiple projections, get a projection list from the projectionList() method on the Projections class. The org.hibernate.criterion.ProjectionList object has an add() method that takes a Projection object. You can pass the projections list to the setProjection() method on the Criteria object because ProjectionList implements the Projection interface.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  ProjectionList projList = Projections.projectionList();  projList.add(Projections.max("price"));  projList.add(Projections.min("price"));  projList.add(Projections.avg("price"));  projList.add(Projections.countDistinct("description"));  crit.setProjection(projList);  List<object[]> results = crit.list(); |
|  |
|  |

**Example 3 : Getting Selected Columns**

Another use of projections is to retrieve individual properties, rather than entities. For instance, we can retrieve just the name and description from our product table, instead of loading the entire object representation into memory.

|  |
| --- |
| Criteria crit = session.createCriteria(Product.class);  ProjectionList projList = Projections.projectionList();  projList.add(Projections.property("name"));  projList.add(Projections.property("description"));  crit.setProjection(projList);  crit.addOrder(Order.asc("price"));  List<object[]> results = crit.list(); |

## Query By Example (QBE)

In QBE, instead of programmatically building a Criteria object with Criterion objects and logical expressions, you can partially populate an instance of the object. You use this instance as a template and have Hibernate build the criteria for you based upon its values. This keeps your code clean and makes your project easier to test.

For instance, if we have a user database, we can construct an instance of a user object, set the property values for type and creation date, and then use the Criteria API to run a QBE query. Hibernate will return a result set containing all user objects that match the property values that were set. Behind the scenes, Hibernate inspects the Example object and constructs an SQL fragment that corresponds to the properties on the Example object.

The following basic example searches for suppliers that match the name on the example Supplier object:

|  |
| --- |
| Criteria crit = session.createCriteria(Supplier.class);  Supplier supplier = new Supplier();  supplier.setName("MegaInc");  crit.add(Example.create(supplier));  List results = crit.list(); |

## Summary

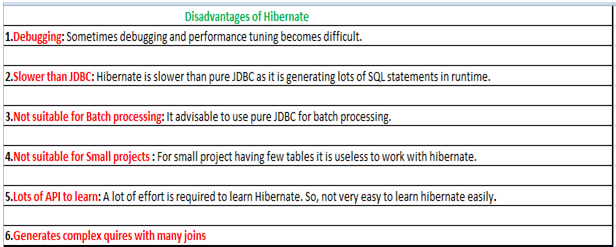
Using the Criteria API is an excellent way to get started developing with HQL. The developers of Hibernate have provided a clean API for adding restrictions to queries with Java objects. Although HQL isn’t too difficult to learn, some developers prefer the Criteria Query API, as it offers compile-time syntax checking—although column names and other schema-dependent information cannot be checked until run time.

# [Advantages and disadvantages of hibernate compared to jdbc](http://www.instanceofjava.com/2016/10/advantages-and-disadvantages-hibernate.html)

Advantages of Hibernate over JDBC:

1. Hibernate is an ORM tool
2. Hibernate is an open source framework.
3. Better than JBDC.
4. Hibernate has an exception translator , which converts checked exceptions of JDBC in to unchecked exceptions of hibernate. So all exceptions in hibernate are unchecked exceptions and Because of this no need to handle exceptions explicitly.
5. Hibernate supports inheritance and polymorphism.
6. With hibernate we can manage the data stored across multiple tables, by applying relations(association)
7. Hibernate has its own query language called Hibernate Query Language. With this HQL hibernate became database independent.
8. Hibernate supports relationships like One-To-One, One-To-Many, Many-To-One ,Many-To-Many.
9. Hibernate has Caching mechanism. using this number of database hits will be reduced. so performance of an application will be increases.
10. Hibernate supports lot of databases.
11. [Hibernate supported databases List](http://www.instanceofjava.com/2016/10/hibernate-supported-databases-list.html).
12. Hibernate is a light weight framework because hibernate uses POJO classes for data transfer between application and database.
13. Hibernate has versioning and time stamp feature with this we can know how many number of times data is modified.
14. Hibernate also supports annotations along with XML.
15. Hibernate supports Lazy loading.
16. Hibernate is easy to learn it is developers friendly.
17. The architecture is layered to keep you isolated from having to know the underlying APIs.
18. Hibernate maintains database connection pool.
19. Hibernate  has Concurrency support.
20. Using Hibernate its Easy to maintain and it will increases productivity

**Disadvantages of Hibernate Compared to JDBC!!:**

* Hibernate is slow compared to JDBC because of generating many sql queries at run time but this is not considered as disadvantage in my view.
* Below are some of the disadvantages but these are not applicable to small applications. But we have given some possible scenarios.

**How to call stored procedure in Hibernate?**

In Hibernate, there are three approaches to call a database store procedure.

1**. Native SQL – createSQLQuery**

You can use **createSQLQuery()** to call a store procedure directly.

Query query = session.createSQLQuery(

"CALL GetStocks(:stockCode)")

.addEntity(Stock.class)

.setParameter("stockCode", "7277");

List result = query.list();

for(int i=0; i<result.size(); i++){

Stock stock = (Stock)result.get(i);

System.out.println(stock.getStockCode());

}

2. **NamedNativeQuery** in annotation

Declare your store procedure inside the **@NamedNativeQueries** annotation.

//Stock.java

...

@NamedNativeQueries({

@NamedNativeQuery(

name = "callStockStoreProcedure",

query = "CALL GetStocks(:stockCode)",

resultClass = Stock.class

)

})

@Entity

@Table(name = "stock")

public class Stock implements java.io.Serializable {

...

Call it with **getNamedQuery()**.

Query query = session.getNamedQuery("callStockStoreProcedure")

.setParameter("stockCode", "7277");

List result = query.list();

for(int i=0; i<result.size(); i++){

Stock stock = (Stock)result.get(i);

System.out.println(stock.getStockCode());

}

3. sql-query in XML mapping file

Declare your store procedure inside the "**sql-query**" tag.

<!-- Stock.hbm.xml -->

...

<hibernate-mapping>

<class name="com.mkyong.common.Stock" table="stock" ...>

<id name="stockId" type="java.lang.Integer">

<column name="STOCK\_ID" />

<generator class="identity" />

</id>

<property name="stockCode" type="string">

<column name="STOCK\_CODE" length="10" not-null="true" unique="true" />

</property>

...

</class>

<sql-query name="callStockStoreProcedure">

<return alias="stock" class="com.mkyong.common.Stock"/>

<![CDATA[CALL GetStocks(:stockCode)]]>

</sql-query>

</hibernate-mapping>

Call it with **getNamedQuery()**.

Query query = session.getNamedQuery("callStockStoreProcedure")

.setParameter("stockCode", "7277");

List result = query.list();

for(int i=0; i<result.size(); i++){

Stock stock = (Stock)result.get(i);

System.out.println(stock.getStockCode());

}

**Conclusion**

The above three approaches are doing the same thing, call a store procedure in database. There are not much big different between the three approaches, which method you choose is depend on your personal prefer.

**Struts2 Hibernate Deployment**

**For web.xml**

**<filter>**

**<filter-name>struts2</filter-name>**

**<filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>**

**</filter>**

**<listener>**

**<listener-class>com.journaldev.struts2hibernate.listener.HibernateServletContextListener</listener-class>**

**</listener>**

**Struts.xml**

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE struts PUBLIC

"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"

"http://struts.apache.org/dtds/struts-2.3.dtd">

<struts>

<constant name="struts.devMode" value="true"></constant>

<constant name="struts.convention.result.path" value="/"></constant>

<package name="user" namespace="/User" extends="struts-default">

<action name="home">

<result>/login.jsp</result>

</action>

<action name="login" class="com.journaldev.struts2hibernate.action.LoginAction">

<result name="success">/welcome.jsp</result>

<result name="error">/error.jsp</result>

</action>

</package>

</struts>

Useful in gaining fast application performance if used correctly. The idea behind cache is to reduce the number of database queries, hence reducing the throughput time of the application.

Hibernate comes with different types of Cache:

# First Level Cache: Hibernate first level cache is associated with the Session object. Hibernate first level cache is enabled by default and there is no way to disable it. However hibernate provides

Hibernate Cache:

Hibernate Cache can be very

1. Methods through which we can delete selected objects from the cache or clear the cache completely. Any object cached in a session will not be visible to other sessions and when the session is closed; all the cached objects will also be lost.
2. Hibernate First Level cache is enabled by default, there are no configurations needed for this.
3. Hibernate first level cache is session specific, that’s why when we are getting the same data in same session there is no query fired whereas in other session query is fired to load the data.
4. Hibernate first level cache can have old values, as you can see above that I have put my program to sleep for 10 seconds and in that time I updated the value (name from Pankaj to PankajK) in database but it didn’t get reflected in the same session. But in other session, we got the updated value.
5. We can use session evict() method to remove a single object from the hibernate first level cache.
6. We can use session clear() method to clear the cache i.e delete all the objects from the cache.
7. We can use session contains() method to check if an object is present in the hibernate cache or not, if the object is found in cache, it returns true or else it returns false.
8. Since hibernate cache all the objects into session first level cache, while running bulk queries or batch updates it’s necessary to clear the cache at certain intervals to avoid memory issues.

**2. Second Level Cache**: Hibernate Second Level cache is disabled by default but we can enable it through configuration. Currently EHCache and Infinite span provides implementation for Hibernate Second level cache and we can use them. We will look into this in the next tutorial for hibernate caching.

One of the major benefit of using Hibernate in large application is it’s support for cache, hence reducing database queries and better performance.

Hibernate Second Level cache providers include EHCache and Infinispan, but EHCache is more popular and we will use it for our example project. However before we move to our project, we should know different strategies for caching an object.

**Read Only**: This caching strategy should be used for persistent objects that will always read but never updated. It’s good for reading and caching application configuration and other static data that are never updated. This is the simplest strategy with best performance because there is no overload to check if the object is updated in database or not.

**Read Write**: It’s good for persistent objects that can be updated by the hibernate application. However if the data is updated either through backend or other applications, then there is no way hibernate will know about it and data might be stale. So while using this strategy, make sure you are using Hibernate API for updating the data.

**Nonrestricted Read Write**: If the application only occasionally needs to update data and strict transaction isolation is not required, a nonstrict-read-write cache might be appropriate.

**Transactional**: The transactional cache strategy provides support for fully transactional cache providers such as JBoss TreeCache. Such a cache can only be used in a JTA environment and you must specify hibernate.transaction.manager\_lookup\_class.

<!-- EHCache Core APIs -->

<dependency>

<groupId>net.sf.ehcache</groupId>

<artifactId>ehcache-core</artifactId>

<version>2.6.9</version>

</dependency>

<!-- Hibernate EHCache API -->

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-ehcache</artifactId>

<version>4.3.5.Final</version>

</dependency>

<!-- EHCache uses slf4j for logging -->

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-simple</artifactId>

<version>1.7.5</version>

</dependency>

<property name="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>

<!-- For [singleton](http://www.journaldev.com/1377/java-singleton-design-pattern-best-practices-examples) factory -->

<!-- <property name="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory</property>

-->

<!-- enable second level cache and query cache -->

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

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

<property name="net.sf.ehcache.configurationResourceName">/myehcache.xml</property>

Some important points about hibernate second level cache configurations are:

1. **hibernate.cache.region.factory\_class** is used to define the Factory class for Second level caching, I am using org.hibernate.cache.ehcache.EhCacheRegionFactory for this. If you want the factory class to be singleton, you should use org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory class.

If you are using Hibernate 3, corresponding classes will be net.sf.ehcache.hibernate.EhCacheRegionFactory and net.sf.ehcache.hibernate.SingletonEhCacheRegionFactory.

1. **hibernate.cache.use\_second\_level\_cache** is used to enable the second level cache.
2. **hibernate.cache.use\_query\_cache** is used to enable the query cache, without it HQL queries results will not be cached.
3. **net.sf.ehcache.configurationResourceName** is used to define the EHCache configuration file location, it’s an optional parameter and if it’s not present EHCache will try to locate **ehcache.xml** file in the application classpath.

Hibernate EHCache provides a lot of options, I won’t go into much detail but some of the important configurations above are:

1. **diskStore**: EHCache stores data into memory but when it starts overflowing, it start writing data into file system. We use this property to define the location where EHCache will write the overflown data.
2. **defaultCache**: It’s a mandatory configuration, it is used when an Object need to be cached and there are no caching regions defined for that.
3. **cache name=”employee”:** We use cache element to define the region and it’s configurations. We can define multiple regions and their properties, while defining model beans cache properties, we can also define region with caching strategies. The cache properties are easy to understand and clear with the name.
4. Cache regions **org.hibernate.cache.internal.StandardQueryCache** and
5. **org.hibernate.cache.spi.UpdateTimestampsCache**are defined because EHCache was giving warning to that.

Hibernate Second Level Cache – Model Bean Caching Strategy

We use **org.hibernate.annotations.Cache** annotation to provide the caching configuration. **org.hibernate.annotations.CacheConcurrencyStrategy** is used to define the caching strategy and we can also define the cache region to use for the model beans.

* As you can see from output, statistics were disabled at first but we enabled it for checking our hibernate second level cache.
* Step by step explanation of the output is as follows:
* Before we load any data in our application, all the stats are 0 as expected.
* When we are loading the Employee with id=1 for the first time, it’s first searched into first level cache and then second level cache. If not found in cache, database query is executed and hence fetch count becomes 1. Once the object is loaded, it’s saved into first level cache and second level cache both. So secondary level hit count remains 0 and miss count becomes 1. Notice that put count is 2, that is because Employee object consists of Address too, so both the objects are saved into second level cache and count is increased to 2.
* Next, we are again loading the employee with id=1, this time it’s present in the first level cache. So you don’t see any database query and all other secondary level cache stats also remains same.
* Next we are using evict() method to remove the employee object from the first level cache, now when we are trying to load it, hibernate finds it in the second level cache. That’s why no database query is fired and fetch count remains 1. Notice that hit count goes from 0 to 2 because both Employee and Address objects are read from the second level cache. Second level miss and put count remains at the earlier value.
* Next we are loading an employee with id=3, database query is executed and fetch count increases to 2, miss count increases from 1 to 2 and put count increases from 2 to 4.
* Next we are trying to load employee with id=1 in another session, Since hibernate second level cache is shared across sessions, it’s found in the second level cache and no database query is executed. Fetch count, miss count and put count remains same whereas hit count increases from 2 to 4.
* So it’s clear that our Hibernate second level cache; Hibernate EHCache; is working fine. Hibernate statistics are helpful in finding the bottleneck in the system and optimize it to reduce the fetch count and load more data from the cache.

**3. Query Cache**: Hibernate can also cache result set of a query. Hibernate Query Cache doesn’t cache the state of the actual entities in the cache; it caches only identifier values and results of value type. So it should always be used in conjunction with the second-level cache.

Most of the times, database tables are associated with each other. There are many forms of association – **one-to-one**, **one-to-many** and **many-to-many** are at the broad level. These can be further divided into unidirectional and bidirectional mappings.

## Hibernate One to One Mapping:

First of all we would need to setup One to One mapping in database tables. We will create two tables for our example – Transaction and Customer. Both of these tables will have one to one mapping. Transaction will be the primary table and we will be using **Foreign Key** in Customer table for one-to-one mapping.

* Without Annotation

For Customer

<id name=*"id"* type=*"long"*>

<column name=*"txn\_id"* />

<generator class=*"foreign"*>

<param name=*"property"*>txn</param>

</generator>

</id>

For Transaction table

<one-to-one name=*"customer"* class=*"com.journaldev.hibernate.model.Customer"*

cascade=*"save-update"* />

<mapping resource=*"txn.hbm.xml"*/>

<mapping resource=*"customer.hbm.xml"*/>

* With Annotation:

For Transaction

@OneToOne(mappedBy="txn")

@Cascade(value=org.hibernate.annotations.CascadeType.***SAVE\_UPDATE***)

**private** Customer1 customer;

For **Customer**

@OneToOne

@PrimaryKeyJoinColumn

**private** Txn1 txn;

@Id

@Column(name="txn\_id", unique=**true**, nullable=**false**)

@GeneratedValue(generator="gen")

@GenericGenerator(name="gen", strategy="foreign", parameters={@Parameter(name="property", value="txn")})

**private** **long** id;

<mapping class=*"com.journaldev.hibernate.model.Txn1"*/>

<mapping class=*"com.journaldev.hibernate.model.Customer1"*/>

## One To Many Mapping in Hibernate

## In simple terms, one to many mapping means that one row in a table can be mapped to multiple rows in another table. For example, think of a Cart system where we have another table for Items. A cart can have multiple items, so here we have one to many mapping.

* Without Annotation:

For Cart

<set name=*"items"* table=*"ITEMS"* fetch=*"select"*>

<key>

<column name=*"cart\_id"* not-null=*"true"*></column>

</key>

<one-to-many class=*"Items"*/>

</set>

## For Items

<many-to-one name=*"cart"* class=*"Cart"*>

<column name=*"cart\_id"* not-null=*"true"*></column>

</many-to-one>

<mapping resource=*"cart.hbm.xml"*/>

<mapping resource=*"items.hbm.xml"*/>

* With Annotation:

For Cart

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

@Column(name="cart\_id")

**private** **long** id;

@OneToMany(mappedBy="cart1")

**private** Set<Items1> items1;

For Items

@Id

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

@Column(name="id")

**private** **long** id;

@ManyToOne

@JoinColumn(name="cart\_id", nullable=**false**)

**private** Cart1 cart1;

<mapping class=*"com.journaldev.hibernate.model.Cart1"*/>

<mapping class=*"com.journaldev.hibernate.model.Items1"*/>

## Hibernate Many to Many

## Many-to-Many mapping is usually implemented in database using a Join Table. For example we can have Cart and Item table and Cart\_Items table for many-to-many mapping. Every cart can have multiple items and every item can be part of multiple carts, so we have a many to many mapping here.

* Without Annotation:

For Cart

<set name=*"items"* table=*"CART\_ITEMS"* fetch=*"select"* cascade=*"all"*>

<key column=*"cart\_id"* />

<many-to-many class=*"Item"* column=*"item\_id"* />

## </set>

## For Items

<set name=*"carts"* table=*"CART\_ITEMS"* fetch=*"select"* cascade=*"all"*>

<key column=*"item\_id"* />

<many-to-many class=*"Cart"* column=*"cart\_id"* />

</set>

<mapping resource=*"cart.hbm.xml"*/>

<mapping resource=*"item.hbm.xml"*/>

* With Annotation:

For Cart

@Id

@Column(name = "cart\_id")

@GeneratedValue(strategy=GenerationType.IDENTITY)

@Column(name="cart\_id")

private long id;

@ManyToMany(targetEntity = Item1.**class**, cascade = { CascadeType.***ALL*** })

@JoinTable(name = "CART\_ITEMS",

joinColumns = { @JoinColumn(name = "cart\_id") },

inverseJoinColumns = { @JoinColumn(name = "item\_id") })

**private** Set<Item1> items;

For Items

@Id

@Column(name="item\_id")

@GeneratedValue(strategy=GenerationType.***IDENTITY***)

**private** **long** id;

<mapping class=*"com.journaldev.hibernate.model.Cart1"*/>

<mapping class=*"com.journaldev.hibernate.model.Item1"*/>

Hibernate Tomcat JNDI DataSource

## Hibernate SQL Query

###### Hibernate provide option to execute native SQL queries through the use of **SQLQuery** object. Hibernate SQL Query is very handy when we have to execute database vendor specific queries that are not supported by Hibernate API. For example query hints or the CONNECT keyword in Oracle Database.

###### For normal scenarios, Hibernate SQL query is not the recommended approach because we loose benefits related to hibernate association and [hibernate first level cache](http://www.journaldev.com/2969/hibernate-caching-first-level-cache).

**Hibernate Native SQL Example**

For Hibernate Native SQL Query, we use Session.createSQLQuery(String query) to create the SQLQuery object and execute it. For example, if you want to read all the records from Employee table, we can do it through below code.

// Prep work

SessionFactory sessionFactory = HibernateUtil.getSessionFactory();

Session session = sessionFactory.getCurrentSession();

// Get All Employees

Transaction tx = session.beginTransaction();

SQLQuery query = session.createSQLQuery("select emp\_id, emp\_name, emp\_salary from Employee");

List<Object[]> rows = query.list();

for(Object[] row : rows){

Employee emp = new Employee();

emp.setId(Long.parseLong(row[0].toString()));

emp.setName(row[1].toString());

emp.setSalary(Double.parseDouble(row[2].toString()));

System.out.println(emp);

}

**Hibernate SQL Query addScalar**

###### Hibernate uses ResultSetMetadata to deduce the type of the columns returned by the query, from performance point of view we can use addScalar() method to define the data type of the column. However we would still get the data in form of Object array. The output generated will be same, however we will see slight performance improvement when the data is huge.

//Get All Employees - addScalar example

query = session.createSQLQuery("select emp\_id, emp\_name, emp\_salary from Employee")

.addScalar("emp\_id", new LongType())

.addScalar("emp\_name", new StringType())

.addScalar("emp\_salary", new DoubleType());

rows = query.list();

for(Object[] row : rows){

Employee emp = new Employee();

emp.setId(Long.parseLong(row[0].toString()));

emp.setName(row[1].toString());

emp.setSalary(Double.parseDouble(row[2].toString()));

System.out.println(emp);

}

**Hibernate Native SQL Multiple Tables**

###### If we would like to get data from both Employee and Address tables, we can simply write the SQL query for that and parse the result set.

query = session.createSQLQuery("select e.emp\_id, emp\_name, emp\_salary,address\_line1, city,

zipcode from Employee e, Address a where a.emp\_id=e.emp\_id");

rows = query.list();

for(Object[] row : rows){

Employee emp = new Employee();

emp.setId(Long.parseLong(row[0].toString()));

emp.setName(row[1].toString());

emp.setSalary(Double.parseDouble(row[2].toString()));

Address address = new Address();

address.setAddressLine1(row[3].toString());

address.setCity(row[4].toString());

address.setZipcode(row[5].toString());

emp.setAddress(address);

System.out.println(emp);

}

**Hibernate Native SQL Entity and Join**

###### We can also use addEntity() and addJoin() methods to fetch the data from associated table using tables join. For example, above data can also be retrieved as below.

//Join example with addEntity and addJoin

query = session.createSQLQuery("select {e.\*}, {a.\*} from Employee e join Address a ON e.emp\_id=a.emp\_id")

.addEntity("e",Employee.class)

.addJoin("a","e.address");

rows = query.list();

for (Object[] row : rows) {

for(Object obj : row) {

System.out.print(obj + "::");

}

System.out.println("\n");

}

//Above join returns both Employee and Address Objects in the array

for (Object[] row : rows) {

Employee e = (Employee) row[0];

System.out.println("Employee Info::"+e);

Address a = (Address) row[1];

System.out.println("Address Info::"+a);

}

###### {[aliasname].\*} is used to return all properties of an entity. When we use addEntity() and addJoin() with join queries like above it returns both the objects, as shown above.

### Hibernate Native SQL Query with Parameters

###### We can also pass parameters to the Hibernate SQL queries, just like [JDBC PreparedStatement](http://www.journaldev.com/2489/jdbc-statement-vs-preparedstatement-sql-injection-example). The parameters can be set using the name as well as index, as shown in below example.

query = session

.createSQLQuery("select emp\_id, emp\_name, emp\_salary from Employee where emp\_id = ?");

List<Object[]> empData = query.setLong(0, 1L).list();

for (Object[] row : empData) {

Employee emp = new Employee();

emp.setId(Long.parseLong(row[0].toString()));

emp.setName(row[1].toString());

emp.setSalary(Double.parseDouble(row[2].toString()));

System.out.println(emp);

}

query = session

.createSQLQuery("select emp\_id, emp\_name, emp\_salary from Employee where emp\_id = :id");

empData = query.setLong("id", 2L).list();

for (Object[] row : empData) {

Employee emp = new Employee();

emp.setId(Long.parseLong(row[0].toString()));

emp.setName(row[1].toString());

emp.setSalary(Double.parseDouble(row[2].toString()));

System.out.println(emp);

}

## Hibernate Log4j Logging

###### This file needs to be placed at the root folder, so that our main class can access it. Notice the location of log4j log files, our application generated logs will go into project.log whereas hibernate logs will go into system.log file.

## Hibernate Validator

###### Welcome to Hibernate Validator Example Tutorial. Data validation is integral part of any application. You will find data validation at presentation layer with the use of Javascript. Then at the server side code before processing the client data. Also data validation occurs before persisting it, to make sure it follows the correct format.

###### Validation is a cross cutting task, so we should try to keep it apart from our business logic. That’s why JSR303 and JSR349 provides specification for validating a bean by using annotations. **Hibernate Validator**provides the reference implementation of both these bean validation specs.

###### It’s very easy to use Hibernate Validator and best part is that we can easily extend it and create our own custom validation annotations.

<dependency>

<groupId>javax.validation</groupId>

<artifactId>validation-api</artifactId>

<version>1.1.0.Final</version>

</dependency>

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-validator</artifactId>

<version>5.1.1.Final</version>

</dependency>

Hibernate Validator also requires an implementation of the Unified Expression Language (JSR 341) for evaluating dynamic expressions in constraint violation messages.

If your application is running in a servlet container such as JBoss, it’s already provided. But if you are using it in a standalone application like my example project, you need to add them manually. Required dependencies are;

<dependency>

<groupId>javax.el</groupId>

<artifactId>javax.el-api</artifactId>

<version>2.2.4</version>

</dependency>

<dependency>

<groupId>org.glassfish.web</groupId>

<artifactId>javax.el</artifactId>

<version>2.2.4</version>

</dependency>

## Hibernate Criteria

###### Most of the times, we use HQL for querying the database and getting the results. HQL is not preferred way for updating or deleting values because then we need to take care of any associations between tables.

###### Hibernate Criteria API provides object oriented approach for querying the database and getting results. We can’t use Criteria in Hibernate to run update or delete queries or any DDL statements. Hibernate Criteria query is only used to fetch the results from the database using object oriented approach.

###### For my Hibernate criteria example, I will use the same setup as in my [HQL Example](http://www.journaldev.com/2954/hibernate-query-language-hql-example-tutorial) and show you how to use Criteria in Hibernate for querying databases.

###### Some of the common usage of Hibernate Criteria API are;

1. Hibernate Criteria API provides Projection that we can use for aggregate functions such as sum(), min(), max() etc.
2. Hibernate Criteria API can be used with ProjectionList to fetch selected columns only.
3. Criteria in Hibernate can be used for join queries by joining multiple tables, useful methods for Hibernate criteria join are createAlias(), setFetchMode() and setProjection()
4. Criteria in Hibernate API can be used for fetching results with conditions, useful methods are add() where we can add Restrictions.
5. Hibernate Criteria API provides addOrder() method that we can use for ordering the results.

# HQL – Hibernate Query Language

###### HQL or Hibernate Query Language is the object-oriented query language of Hibernate Framework. HQL is very similar to SQL except that we use Objects instead of table names that makes it more close to object oriented programming.

**HQL and Case Sensitivity**: HQL is case-insensitive except for java class and variable names. So SeLeCT is the same as sELEct is the same as SELECT, but com.journaldev.model.Employee is not same as com.journaldev.model.EMPLOYEE.

Some of the commonly supported clauses in HQL are:

1. **HQL From**: HQL From is same as select clause in SQL, from Employee is same as select \* from Employee. We can also create alias such as from Employee emp or from Employee as emp.
2. **HQL Join**: HQL supports inner join, left outer join, right outer join and full join. For example, select e.name, a.city from Employee e INNER JOIN e.address a. In this query, Employee class should have a variable named address. We will look into it in the example code.
3. **Aggregate Functions**: HQL supports commonly used aggregate functions such as count(\*), count(distinct x), min(), max(), avg() and sum().
4. **Expressions**: HQL supports arithmetic expressions (+, -, \*, /), binary comparison operators (=, >=, <=, <>, !=, like), logical operations (and, or, not) etc.
5. HQL also supports ordre by and group by clauses.
6. HQL also supports sub-queries just like SQL queries.
7. HQL supports DDL, DML and executing store procedures too

# Pagination Example Hibernate

###### If Hibernate fetches large amount of data (records) from the database, it consumes lot of memory. To consume less RAM memory, the data can be obtained in installments from the database. One solution is using pagination

.

The idea behind Pagination Hibernate is to divide the large result set into a number of pages and fetching one page at a time. We can programmatically declare how many records should contain each page and from what record. For example, the page may contain 5 records staring from 3rd record.

The code is simple to do the job of Pagination Hibernate.

Think there are 7 records (counting 0 to 6) in **school** table. We make three pages where 2 records, 3 records and 3 records are fetched each time (three database hits are made). End fetching gets only 2 records.

**Note 1:** Observe, for each fetch, one database hit is made. First fetch 2 records, second fetch 3 records and last fetch 2 records are obtained.

Note 2: It saves time also (or increases performance); for a number of records (one page) only one database hit is made.

|  |  |
| --- | --- |
|  | Query q = session.createQuery("select \* from Student");  // you can use order by also  q.setFirstResult(3);    // starting position of the record (first record is 0, that is, 0, 1, 2, 3)  q.setMaxResults(5);           // iterate the list to get each page with Iterator or for loop |

   // size of page; each page displays 5 records (3, 4, 5, 6, 7)

List list1 = q.list();

**Hibernate vs iBATIS - Comparison of two powerful ORM tools**

Hibernate and iBatis both are open source Object Relational Mapping (ORM) tools available in the industry. Both of these tools are very powerful and use of each of these depends on the context. Below are few criteria we can keep in our mind while choosing one over other. Within the java persistence there is no one size, fits all solution. Hibernate is most commonly used ORM tool.

1. **Way of working**

Hibernate maps your Java based POJO objects to corresponding Database tables where as iBatis maps the ResultSet from JDBC API to Java POJO Objets. So in case of iBatis you may need to pass your parameters to the DAO class in a map object which can be passed to the Statement.  
2. **XML mappings**

Hibernate and iBatis both use XML based mapping and are quite intuitive to implement. Tools are available to create and edit these file. e.g. [Abator for iBatis](http://ibatis.apache.org/docs/tools/abator/) and [Hibernate tools](http://www.hibernate.org/255.html).  
3. **Project from Scratch with new Database or Existing Database**

Hibernate works well when you control the data model, iBATIS works well when you need to integrate with an existing database. In the projects where the Project is being developed from scratch and developer has to come up with database design along with the Java object model then Hibernate is a good option as it becomes easy to develop and mappings becomes intuitive. Hibernate also provides ways to generate database from mappings files itself. Which may reduce your deployment script generation efforts.  
 Using iBatis for a project which is developed from scratch is also very simple and does not involve lot of efforts.  
4. **Data centric project**

A project that needs database intensive operations and needs to use lot of existing implementations that are available in Database side then iBatis provides simple ways to leverage it. Hibernate also provides way to handle such situations but it gets complex as your application becomes more and more complex. The hibernate developer need to start understanding lot of complex things to accomplish it.  
5. **Hibernate Query Language (HQL) vs. Structured Query Language (SQL)**

Its another choice which you may want to take in case your application involves lot of complex database intensive operations then in Hibernate you may want to go for Hibernate Query Language (HQL) which will have following aspects Learning curve for developers

You will not be able to see the actual SQL because Hibernate generates it internally, though there is a way to see what is being generated but you lose the direct control over it.

- Abstraction of SQL makes it difficult for the developers visualize what is happening at that layer. Where as in case of iBatis you have same SQL available to you in the XML mapping file and you can play around with it as you wish, which gives you lot of freedom to tune in performance of queries.

6. **Simple Applications**

In case your application is simple with not lot of complex data processing then Hibernate could be a decent option. Also if a developer does not like SQL much then it could be a good option too.

7. **SPRING Integration**

Hibernate and iBatis both have good support from SPRING framework so it should not be a problem to choose one of them.

**Session.load() vs Session.get()**

**session.load()**

It will always return a “proxy” (Hibernate term) without hitting the database. In Hibernate, proxy is an object with the given identifier value, its properties are not initialized yet, it just look like a temporary fake object.

If no row found, it will throws an ObjectNotFoundException.

**session.get()**

It always hit the database and return the real object, an object that represent the database row, not proxy.

If no row found , it return null.

Hibernate create anything for some reasons, when you do the association, it’s normal to obtain retrieve an object (persistent instance) from database and assign it as a reference to another object, just to maintain the relationship. Let’s go through some examples to understand in what situation you should use **session.load().**

**session.load()**

In above scenario, **session.load()** will be your good solution, let’s see the example,

Stock stock = (Stock)session.load(Stock.class, new Integer(2));

StockTransaction stockTransactions = new StockTransaction();

//set stockTransactions detail

stockTransactions.setStock(stock);

session.save(stockTransactions);

Output

Hibernate:

insert into mkyong.stock\_transaction (...) values (?, ?, ?, ?, ?, ?)

In session.load(), Hibernate will not hit the database (no select statement in output) to retrieve the Stock object, it will return a Stock proxy object – a fake object with given identify value. In this scenario, a proxy object is enough for to save a stock transaction record.

Exception

In exception case, see the examples

**session.load()**

Stock stock = (Stock)session.load(Stock.class, new Integer(100)); //proxy

//initialize proxy, no row for id 100, throw ObjectNotFoundException

System.out.println(stock.getStockCode());

It will always return a proxy object with the given identity value, even the identity value is not exists in database. However, when you try to initialize a proxy by retrieve it’s properties from database, it will hit the database with select statement. If no row is found, a **ObjectNotFoundException** will throw.

org.hibernate.ObjectNotFoundException: No row with the given identifier exists:

[com.mkyong.common.Stock#100]

**session.get()**

//return null if not found

Stock stock = (Stock)session.get(Stock.class, new Integer(100));

System.out.println(stock.getStockCode()); //java.lang.NullPointerException

It will always return null , if the identity value is not found in database

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

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

Second difference is that we can use persist () method only within the boundary of a transaction, so it’s safe and takes care of any cascaded objects.

**Hibernate *saveOrUpdate*** results into insert or update queries based on the provided data. If the data is present in the database, update query is executed. We can use saveOrUpdate without transaction also, but again you will face the issues with mapped objects not getting saved if session is not flushed. Hibernate saveOrUpdate adds the entity object to persistent context and track any further changes. Any further changes are saved at the time of committing transaction, like persist.

**Hibernate update** should be used where we know that we are only updating the entity information. This operation adds the entity object to persistent context and further changes are tracked and saved when transaction is committed.

**Hibernate merge** can be used to update existing values, however this method create a copy from the passed entity object and return it. The returned object is part of persistent context and tracked for any changes, passed object is not tracked.

**What is Lazy/Select Fetch / Eagar Fetch in Hibernate?**

***Lazy Fetching****:* This is the default Fetch type of **hibernate3**. Select fetch strategy is the lazy fetching of associations. The purpose of lazy strategy is memory optimization. When I say memory optimization that means it saves us from heap error. This is what think. So we can say yes if we are loading too objects in a session we should go for lazy fetching strategy but in terms of time performance it doesn’t provide any benefit.

***Eagar/Join Fetching****: In hibernate 2, this is the default of retrieving object from the database.*

*The purpose* of join fetch strategy is optimization in terms of time. I mean associations are fetched right at the time of fetching parent object. In this case we don’t make database call again and again. So, this will be much faster. Agreed that this will bad if we are fetching too many objects in session because we can get java heap error.

**What is cascading and what are different types of cascading?**

When we have relationship between entities, then we need to define how the different operations will affect the other entity. This is done by cascading and there are different types of it.

Here is a simple example of applying cascading between primary and secondary entities.

import org.hibernate.annotations.Cascade;

@Entity

.

.

@Table(name = "EMPLOYEE")

public class Employee {

s

@OneToOne(mappedBy = "employee")

@Cascade(value = org.hibernate.annotations.CascadeType.ALL)

private Address address;

}

Note that Hibernate CascadeType enum constants are little bit different from JPA javax.persistence.CascadeType, so we need to use the Hibernate CascadeType and Cascade annotations for mappings, as shown in above example.  
Commonly used cascading types as defined in CascadeType enum are:

**None**: No Cascading, it’s not a type but when we don’t define any cascading then no operations in parent affects the child.

ALL: Cascades save, delete, update, evict, lock, replicate, merge, and persist. Basically everything

SAVE\_UPDATE: Cascades save and update, available only in hibernate.

DELETE: Corresponds to the Hibernate native DELETE action, only in hibernate.

DETATCH, MERGE, PERSIST, REFRESH and REMOVE – for similar operations

LOCK: Corresponds to the Hibernate native LOCK action.

REPLICATE: Corresponds to the Hibernate native REPLICATE action.

**Which design patterns are used in Hibernate framework?**

Some of the design patterns used in Hibernate Framework are:

**Domain Model Pattern** – An object model of the domain that incorporates both behavior and data.

**Data Mapper** – A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.

[**Proxy Pattern**](http://www.journaldev.com/1572/proxy-design-pattern) for lazy loading

[**Factory pattern**](http://www.journaldev.com/1392/factory-design-pattern-in-java)in SessionFactory

**What are the Core interfaces of Hibernate Framework?**

Session, SessionFactory, Configuration, Transaction,

Query and Criteria interfaces.

**How Hibernate is database independent explain?**

Only changing the property in XML

<property name=”hibernate.dialect”>org.hibernate.dialect.Oracle9Dialect</property>

<property name=”hibernate.connection.driver\_class”>oracle.jdbc.driver.OracleDriver</property>

**What is the hibernate proxy?**

An object is just a way to avoid retrieving an object until you need it.

Hibernate2 does not proxy objects by default.

**How do you create a session factory in hibernate?**

Configuration cfg=new Configuration();

cfg.addResource(“dir/hibernate.hbm.xml”);

cfg.setProperties(System.getProperties());

SessionFactory sessions=cfg.buildSessionFactory();

**What is a Thin Client?**

A thin client is a program interface to the application that does not have any operation like query of databases, execute complex business rules, or connect to legacy applications.

**What are the key benifits of Hibernate?**

There are several benefits of using Hibernate

Powerful object-oriented hibernate query language

Transparent persistence based on POJOs without byte code processing

Descriptive O/R Mapping through mapping file.

Automatic primary key generation

Hibernate cache : Session Level, Query and Second level cache.

Performance: Lazy initialization, Outer join fetching, Batch fetching

**What is hibernate session and session factory? How do you configure sessionfactory in spring configuration file?**

Hibernate Session is the main runtime interface between a Java application and Hibernate. SessionFactory allows applications to create hibernate session by reading hibernate configurations file hibernate.cfg.xml.

    Configuration cfg = new Configuration().configure();

// Initialize the Hibernate environment

    SessionFactory factory = cfg.buildSessionFactory();

// Create the session factory

    Session session = factory.openSession();

 // Obtain the new session object

The call to Configuration().configure() loads the hibernate.cfg.xml configuration file and initializes the Hibernate environment. Once the configuration is initialized, you can make any additional modifications you desire programmatically. However, you must make these modifications prior to creating the SessionFactory instance. An instance of SessionFactory is typically created once and used to create all sessions related to a given context.

The main function of the Session is to offer create, read and delete operations for instances of mapped entity classes. Instances may exist in one of three states:

**transient**: never persistent, not associated with any Session

**persistent**: associated with a unique Session

**detached**: previously persistent, not associated with any Session

A Hibernate Session object represents a single unit-of-work for a given data store and is opened by a SessionFactory instance. You must close Sessions when all work for a transaction is completed. The following illustrates a typical Hibernate session:

    Session session = null;

    UserInfo user = null;

    Transaction tx = null;

    try {

       session = factory.openSession();

       tx = session.beginTransaction();

       user = (UserInfo)session.load(UserInfo.class, id);

       tx.commit();

    } catch(Exception e) {

       if (tx != null) {

          try {

             tx.rollback();

          } catch (HibernateException e1) {

             throw new DAOException(e1.toString()); }

       } throw new DAOException(e.toString());

    } finally {

       if (session != null) {

          try {

             session.close();

          } catch (HibernateException e) { }

       }

    }

**What is the difference between hibernate get and load methods?**

The following Hibernate code snippet retrieves a User object from the database:

    User user = (User) session.get(User.class, userID);

The get() method is special because the identifier uniquely identifies a single instance of a class. Hence it’s common for applications to use the identifier as a convenient handle to a persistent object. Retrieval by identifier can use the cache when retrieving an object, avoiding a database hit if the object is already cached. The get() method returns null if the object can’t be found.

Hibernate also provides a load() method:

    User user = (User) session.load(User.class, userID);

If load() can’t find the object in the cache or database, an exception is thrown. The load() method never returns null.  The load() method may return a proxy instead of a real persistent instance. A proxy is a placeholder instance of a runtime-generated subclass (through cglib or Javassist) of a mapped persistent class, it can initialize itself if any method is called that is not the mapped database identifier getter-method.

On the other hand, get() never returns a proxy. Choosing between get() and load() is easy: If you’re certain the persistent object exists, and nonexistence would be considered exceptional, load() is a good option. If you aren’t certain there is a persistent instance with the given identifier, use get() and test the return value to see if it’s null. Using load() has a further implication: The application may retrieve a valid reference (a proxy) to a persistent instance without hitting the database to retrieve its persistent state. So load() might not throw an exception when it doesn’t find the persistent object in the cache or database; the exception would be thrown later, when the proxy is accessed.

**What type of transaction management is supported in hibernate?**

Hibernate communicates with the database via a JDBC Connection; hence it must support both managed and non-managed transactions.

Non-managed in web containers managed in application server using JTA:

    <bean id="transactionManager" class="org.springframework.orm.hibernate.HibernateTransactionManager">

        <property name="sessionFactory">

            <ref local="sessionFactory"/>

        </property>

    </bean>

    <bean id="transactionManager" class="org.springframework.transaction.jta.JtaTransactionManager.">

        <property name="sessionFactory">

            <ref local="sessionFactory"/>

        </property>

    </bean>

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

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

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

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

**What are the different fetching strategy in Hibernate?**

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.

**What are different types of cache hibernate supports ?**

Caching is widely used for optimizing database applications. Hibernate uses two different caches for objects:

**First-level cache**: First-level cache is associated with the Session objectBy default, Hibernate uses first-level cache on a per-transaction basis. Hibernate uses this cache mainly to reduce the number of SQL queries it needs to generate within a given transaction. For example, if an object is modified several times within the same transaction, Hibernate will generate only one SQL UPDATE statement at the end of the transaction, containing all the modifications.

**Second-level cache**: second-level cache is associated with the Session Factory object. To reduce database traffic, second-level cache keeps loaded objects at the Session Factory level between transactions. These objects are available to the whole application, not just to the user running the query. This way, each time a query returns an object that is already loaded in the cache, one or more database transactions potentially are avoided.

**Query-level cache**: In addition, you can use a query-level cache if you need to cache actual query results, rather than just persistent objects. The query cache should always be used in conjunction with the second-level cache. Hibernate supports the following open-source cache implementations out-of-the-box:

**EHCache** is a fast, lightweight, and easy-to-use in-process cache. It supports read-only and read/write caching, and memory- and disk-based caching. However, it does not support clustering.

**OSCache** is another open-source caching solution. It is part of a larger package, which also provides caching functionalities for JSP pages or arbitrary objects. It is a powerful and flexible package, which, like EHCache, supports read-only and read/write caching, and memory- and disk-based caching. It also provides basic support for clustering via either JavaGroups or JMS.

**SwarmCache** is a simple cluster-based caching solution based on JavaGroups. It supports read-only or nonstrict read/write caching (the next section explains this term). This type of cache is appropriate for applications that typically have many more read operations than write operations.

JBoss TreeCache is a powerful replicated (synchronous or asynchronous) and transactional cache. Use this solution if you really need a true transaction-capable caching architecture.

**Commercial Tangosol Coherence cache**.

## CONCURRENCY STRATEGIES:

A concurrency strategy is a mediator which responsible for storing items of data in the cache and retrieving them from the cache. If you are going to enable a second-level cache, you will have to decide, for each persistent class and collection, which cache concurrency strategy to use.

1. Transactional: Use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions,in the rare case of an update.
2. Read-write: Again use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions,in the rare case of an update.
3. Nonstrict-read-write: This strategy makes no guarantee of consistency between the cache and the database. Use this strategy if data hardly ever changes and a small likelihood of stale data is not of critical concern.
4. Read-only: A concurrency strategy suitable for data which never changes. Use it for reference data only.

**What are the different caching strategies?**

**Read-only**: This strategy is useful for data that is read frequently but never updated. This is by far the simplest and best-performing cache strategy.

**Read/write**: Read/write caches may be appropriate if your data needs to be updated. They carry more overhead than read-only caches. In non-JTA environments, each transaction should be completed when Session.close() or Session.disconnect() is called.

**Nonstrict read/write**: This strategy does not guarantee that two transactions won't simultaneously modify the same data. Therefore, it may be most appropriate for data that is read often but only occasionally modified.

**Transactional**: This is a fully transactional cache that may be used only in a JTA environment.

**How do you configure 2nd level cach in hibernate?**

To activate second-level caching, you need to define the hibernate.cache.provider\_class property in the hibernate.cfg.xml file as follows:

    <hibernate-configuration>

        <session-factory>

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

        </session-factory>

    </hibernate-configuration>

By default, the second-level cache is activated and uses the EHCache provider.

To use the query cache you must first enable it by setting the property hibernate.cache.use\_query\_cache to true in hibernate.properties.

**What is the difference between sorted and ordered collection in hibernate?**

A sorted collection is sorted in-memory using java comparator, while order collection is ordered at the database level using order by clause.

**What are the types of inheritence models and describe how they work like vertical inheritence and horizontal?**

There are three types of inheritance mapping in hibernate:

Example: Let us take the simple example of 3 java classes. Class Manager and Worker are inherited from Employee Abstract class.

**Table per concrete class with unions**: In this case there will be 2 tables. Tables: Manager, Worker [all common attributes will be duplicated]

In Single table per subclass, the union of all the properties from the inheritance hierarchy is mapped to one table. As all the data goes in one table, a discriminator is used to differentiate between different type of data.  
**Advantages of Single Table per class hierarchy**

* Simplest to implement.
* Only one table to deal with.
* Performance wise better than all strategies because no joins or sub-selects need to be performed.

**Disadvantages:**

* Most of the column of table are nullable so the NOT NULL constraint cannot be applied.
* Tables are not normalized.

For user convenience we can override the default value of column as well as column name by using the following annotation.  
**@DiscriminatorColumn**  
Target:  
  Classes  
  
Specifies the discriminator column for the SINGLE\_TABLE and JOINED Inheritance mapping strategies.  
The strategy and the discriminator column are only specified in the root of an entity class hierarchy or subhierarchy in which a different inheritance strategy is applied   
If the DiscriminatorColumn annotation is missing, and a discriminator column is required, the name of the discriminator column defaults to "DTYPE" and the discriminator type to DiscriminatorType.STRING.  
 **@DiscriminatorValue**  
Target:  
  Classes  
  
Specifies the value of the discriminator column for entities of the given type.  
  
The DiscriminatorValue annotation can only be specified on a concrete entity class.  
  
If the DiscriminatorValue annotation is not specified and a discriminator column is used, a provider-specific function will be used to generate a value representing the entity type. If the DiscriminatorType is STRING, the discriminator value default is the entity name.  
  
The inheritance strategy and the discriminator column are only specified in the root of an entity class hierarchy or subhierarchy in which a different inheritance strategy is applied. The discriminator value, if not defaulted, should be specified for each entity class in the hierarchy.  
 **@Inheritance**  
  
Target:   
  Classes  
  
Defines the inheritance strategy to be used for an entity class hierarchy. It is specified on the entity class that is the root of the entity class hierarchy. If the Inheritance annotation is not specified or if no inheritance type is specified for an entity class hierarchy, the SINGLE\_TABLE mapping strategy is used.  
  
Now adding the following annotation to the Vehicle class is

1. @Entity
2. @Table(name="VEHICLE")
3. @Inheritance(strategy=InheritanceType.SINGLE\_TABLE) //Least normalisation strategy
4. @DiscriminatorColumn(
5. name="VEHICLE\_TYPE",
6. discriminatorType=DiscriminatorType.STRING
7. )
8. **public** **class** Vehicle
9. {

Now adding following annotation to the TwoWheeler class

1. @DiscriminatorValue("Bike")
2. **public** **class** TwoWheeler **extends** Vehicle
3. {

Now adding following annotation to the FourWheeler class

1. @DiscriminatorValue("Car")
2. **public** **class** FourWheeler **extends** Vehicle
3. {

**Table per class hierarchy:** Single Table can be mapped to a class hierarchy. There will be only one table in database called 'Employee' that will represent all the attributes required for all 3 classes. But it needs some discriminating column to differentiate between Manager and worker;

In this case every entity class has its own table i.e. table per class. The data for Vehicle is duplicated in both the tables.  
This strategy is not popular and also have been made optional in Java Persistence API.  
Advantage:

Possible to define NOT NULL constraints on the table.

Disadvantage:  
 Tables are not normalized.

To support polymorphism either container has to do multiple trips to database or use SQL UNION kind of feature.

In this case there no need for the discriminator column because all entity has own table.

**Table per subclass**: In this case there will be 3 tables represent Employee, Manager and Worker.

It's highly normalized but performance is not good.  
Advantage:

* Tables are normalized.
* Able to define NOT NULL constraint.

Disadvantage:

* Does not perform as well as SINGLE\_TABLE strategy

Using Join Strategy with the vehicle entity

|  |  |  |  |
| --- | --- | --- | --- |
| **Criteria** | **Single Table** | **Table per subclass(Join Strategy)** | **Table per Class** |
| **Table Support** | * Data not normalized. * Constraint for mandatory columns to be not nullable cannot apply. * Change in any subclass leads to change in structure of Table | * Normalized. * Mandatory column constraint can be applied | * One table for each concrete class. * Not maintainable. * Change in base class leads to changes in all tables of derived class |
| **Discriminator Column** | Present | Absent | Absent |
| **Retrieving data** | simple SELECT. All data is in one table. Using discriminator type, individual types can be selected | Joins among table. For example fetching FourWheeler will require a join on FourWheeler and Vehicle table. If all user needs to be fetched than it will put a join for all three tables | Separate Select or Union Select |
| **Updating and Inserting** | Single INSERT or UPDATE | Multiple. For Vehicle type one insert on Vehicle table. For FourWheeler type one insert on Vehicle table and another on FourWheeler table. | One insert or update for each subclass |
| **JPA Support** | Mandatory | Optional |  |

**What is the difference between the session.get() method and the session.load() method?**

Both the session.get(..) and session.load() methods create a persistent object by loading the required object from the database. But if there was not such object in the database then the method session.load(..) throws an exception whereas session.get(&) returns null.

**What is the difference between the session.update() method and the session.lock() method?**

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.

Note: When you reattach detached objects you need to make sure that the dependent objects are reatched as well.

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

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

**What are the general considerations or best practices for defining your Hibernate persistent classes?**

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.

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.

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

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

Use XDoclet tags for generating your \*.hbm.xml files or Annotations (JDK 1.5 onwards), which are less verbose than \*.hbm.xml files.

**How will you configure Hibernate?**

The configuration files hibernate.cfg.xml (or hibernate.properties) and mapping files \*.hbm.xml are used by the Configuration class to create (i.e. configure and bootstrap hibernate) the SessionFactory, which in turn creates the Session instances. Session instances are the primary interface for the persistence service.   
  
" hibernate.cfg.xml (alternatively can use hibernate.properties): These two files are used to configure the hibernate sevice (connection driver class, connection URL, connection username, connection password, dialect etc). If both files are present in the classpath then hibernate.cfg.xml file overrides the settings found in the hibernate.properties file.   
  
" Mapping files (\*.hbm.xml): These files are used to map persistent objects to a relational database. It is the best practice to store each object in an individual mapping file (i.e mapping file per class) because storing large number of persistent classes into one mapping file can be difficult to manage and maintain. The naming convention is to use the same name as the persistent (POJO) class name. For example Account.class will have a mapping file named Account.hbm.xml. Alternatively hibernate annotations can be used as part of your persistent class code instead of the \*.hbm.xml files.   
  
**What is a SessionFactory? Is it a thread-safe object?**  
  
SessionFactory is Hibernates concept of a single datastore and is threadsafe so that many threads can access it concurrently and request for sessions and immutable cache of compiled mappings for a single database. A SessionFactory is usually only built once at startup. SessionFactory should be wrapped in some kind of singleton so that it can be easily accessed in an application code.   
  
SessionFactory sessionFactory = new Configuration().configure().buildSessionfactory();   
  
**What is a 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;   
}   
}   
  
It is also vital that you close your session after your unit of work completes. Note: Keep your Hibernate Session API handy.   
  
**What are the benefits of detached objects?**   
  
 Detached objects can be passed across layers all the way up to the presentation layer without having to use any DTOs (Data Transfer Objects). You can later on re-attach the detached objects to another session.   
  
**What are the pros and cons of detached objects?**   
   
Pros:

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

Listed below is the comparison chart of difference beetween JDBC and Hibernate

|  |  |
| --- | --- |
| JDBC | Hibernate |
| With JDBC, developer has to write code to map an object model's data representation to a relational data model and its corresponding database schema. | Hibernate is flexible and powerful ORM solution to map Java classes to database tables. Hibernate itself takes care of this mapping using XML files so developer does not need to write code for this. |
| With JDBC, the automatic mapping of Java objects with database tables and vice versa conversion is to be taken care of by the developer manually with lines of code. | Hibernate provides transparent persistence and developer does not need to write code explicitly to map database tables tuples to application objects during interaction with RDBMS. |
| JDBC supports only native Structured Query Language (SQL). Developer has to find out the efficient way to access database, i.e. to select effective query from a number of queries to perform same task. | Hibernate provides a powerful query language Hibernate Query Language (independent from type of database) that is expressed in a familiar SQL like syntax and includes full support for polymorphic queries. Hibernate also supports native SQL statements. It also selects an effective way to perform a database manipulation task for an application. |
| Application using JDBC to handle persistent data (database tables) having database specific code in large amount. The code written to map table data to application objects and vice versa is actually to map table fields to object properties. As table changed or database changed then it’s essential to change object structure as well as to change code written to map table-to-object/object-to-table. | Hibernate provides this mapping itself. The actual mapping between tables and application objects is done in XML files. If there is change in Database or in any table then the only need to change XML file properties. |
| With JDBC, it is developer’s responsibility to handle JDBC result set and convert it to Java objects through code to use this persistent data in application. So with JDBC, mapping between Java objects and database tables is done manually. | Hibernate reduces lines of code by maintaining object-table mapping itself and returns result to application in form of Java objects. It relieves programmer from manual handling of persistent data, hence reducing the development time and maintenance cost. |
| With JDBC, caching is maintained by hand-coding. | Hibernate, with Transparent Persistence, cache is set to application work space. Relational tuples are moved to this cache as a result of query. It improves performance if client application reads same data many times for same write. Automatic Transparent Persistence allows the developer to concentrate more on business logic rather than this application code. |
| In JDBC there is no check that always every user has updated data. This check has to be added by the developer. | Hibernate enables developer to define version type field to application, due to this defined field Hibernate updates version field of database table every time relational tuple is updated in form of Java class object to that table. So if two users retrieve same tuple and then modify it and one user save this modified tuple to database, version is automatically updated for this tuple by Hibernate. When other user tries to save updated tuple to database then it does not allow saving it because this user does not have updated data. |

Hibernate Interview Questions and Answers

What is Hibernate Framework?

**Object-relational mapping** or ORM is the programming technique to map application domain model objects to the relational database tables. Hibernate is java based ORM tool that provides framework for mapping application domain objects to the relational database tables and vice versa.

Hibernate provides reference implementation of Java Persistence API, that makes it a great choice as ORM tool with benefits of loose coupling. We can use Hibernate persistence API for CRUD operations. Hibernate framework provide option to map plain old java objects to traditional database tables with the use of JPA annotations as well as XML based configuration.

Similarly hibernate configurations are flexible and can be done from XML configuration file as well as programmatically. For a quick overview of hibernate framework usage, you can go through [Hibernate Beginners Tutorial](http://www.journaldev.com/2882/hibernate-tutorial-for-beginners).

What is Java Persistence API (JPA)?

Java Persistence API (JPA) provides specification for managing the relational data in applications. Current JPA version 2.1 was started in July 2011 as JSR 338. JPA 2.1 was approved as final on 22 May 2013.

JPA specifications is defined with annotations in javax.persistence package. Using JPA annotation helps us in writing implementation independent code.

What are the important benefits of using Hibernate Framework?

Some of the important benefits of using hibernate framework are:

Hibernate eliminates all the boiler-plate code that comes with JDBC and takes care of managing resources, so we can focus on business logic.

Hibernate framework provides support for XML as well as JPA annotations, that makes our code implementation independent.

Hibernate provides a powerful query language (HQL) that is similar to SQL. However, HQL is fully object-oriented and understands concepts like inheritance, polymorphism and association.

Hibernate is an open source project from Red Hat Community and used worldwide. This makes it a better choice than others because learning curve is small and there are tons of online documentations and help is easily available in forums.

Hibernate is easy to integrate with other Java EE frameworks, it’s so popular that Spring Framework provides built-in support for integrating hibernate with Spring applications.

Hibernate supports lazy initialization using proxy objects and perform actual database queries only when it’s required.

Hibernate cache helps us in getting better performance.

For database vendor specific feature, hibernate is suitable because we can also execute native sql queries.

Overall hibernate is the best choice in current market for ORM tool, it contains all the features that you will ever need in an ORM tool.

What are the advantages of Hibernate over JDBC?

Some of the important advantages of Hibernate framework over JDBC are:

Hibernate removes a lot of boiler-plate code that comes with JDBC API, the code looks more cleaner and readable.

Hibernate supports inheritance, associations and collections. These features are not present with JDBC API.

Hibernate implicitly provides transaction management, in fact most of the queries can’t be executed outside transaction. In JDBC API, we need to write code for transaction management using commit and rollback. Read more at [JDBC Transaction Management](http://www.journaldev.com/2483/java-jdbc-transaction-management-savepoint).

JDBC API throws SQLException that is a checked exception, so we need to write a lot of try-catch block code. Most of the times it’s redundant in every JDBC call and used for transaction management. Hibernate wraps JDBC exceptions and throw JDBCException or HibernateException un-checked exception, so we don’t need to write code to handle it. Hibernate built-in transaction management removes the usage of try-catch blocks.

Hibernate Query Language (HQL) is more object oriented and close to java programming language. For JDBC, we need to write native sql queries.

Hibernate supports caching that is better for performance, JDBC queries are not cached hence performance is low.

Hibernate provide option through which we can create database tables too, for JDBC tables must exist in the database.

Hibernate configuration helps us in using JDBC like connection as well as JNDI DataSource for connection pool. This is very important feature in enterprise application and completely missing in JDBC API.

Hibernate supports JPA annotations, so code is independent of implementation and easily replaceable with other ORM tools. JDBC code is very tightly coupled with the application.

Name some important interfaces of Hibernate framework?

Some of the important interfaces of Hibernate framework are:

**SessionFactory (org.hibernate.SessionFactory)**: SessionFactory is an immutable thread-safe cache of compiled mappings for a single database. We need to initialize SessionFactory once and then we can cache and reuse it. SessionFactory instance is used to get the Session objects for database operations.

**Session (org.hibernate.Session)**: Session is a single-threaded, short-lived object representing a conversation between the application and the persistent store. It wraps JDBC java.sql.Connection and works as a factory for org.hibernate.Transaction. We should open session only when it’s required and close it as soon as we are done using it. Session object is the interface between java application code and hibernate framework and provide methods for CRUD operations.

**Transaction (org.hibernate.Transaction)**: Transaction is a single-threaded, short-lived object used by the application to specify atomic units of work. It abstracts the application from the underlying JDBC or JTA transaction. A org.hibernate.Session might span multiple org.hibernate.Transaction in some cases.

What is hibernate configuration file?

Hibernate configuration file contains database specific configurations and used to initialize SessionFactory. We provide database credentials or JNDI resource information in the hibernate configuration xml file. Some other important parts of hibernate configuration file is Dialect information, so that hibernate knows the database type and mapping file or class details.

What is hibernate mapping file?

Hibernate mapping file is used to define the entity bean fields and database table column mappings. We know that JPA annotations can be used for mapping but sometimes XML mapping file comes handy when we are using third party classes and we can’t use annotations.

Name some important annotations used for Hibernate mapping?

Hibernate supports JPA annotations and it has some other annotations in org.hibernate.annotations package. Some of the important JPA and hibernate annotations used are:

**javax.persistence.Entity**: Used with model classes to specify that they are entity beans.

**javax.persistence.Table**: Used with entity beans to define the corresponding table name in database.

**javax.persistence.Access**: Used to define the access type, either field or property. Default value is field and if you want hibernate to use getter/setter methods then you need to set it to property.

**javax.persistence.Id**: Used to define the primary key in the entity bean.

**javax.persistence.EmbeddedId**: Used to define composite primary key in the entity bean.

**javax.persistence.Column**: Used to define the column name in database table.

**javax.persistence.GeneratedValue**: Used to define the strategy to be used for generation of primary key. Used in conjunction with javax.persistence.GenerationType enum.

**javax.persistence.OneToOne**: Used to define the one-to-one mapping between two entity beans. We have other similar annotations as OneToMany, ManyToOne and ManyToMany

**org.hibernate.annotations.Cascade**: Used to define the cascading between two entity beans, used with mappings. It works in conjunction with org.hibernate.annotations.CascadeType

**javax.persistence.PrimaryKeyJoinColumn**: Used to define the property for foreign key. Used with org.hibernate.annotations.GenericGenerator and org.hibernate.annotations.Parameter

Here are two classes showing usage of these annotations.

package com.journaldev.hibernate.model;

import javax.persistence.Access;

import javax.persistence.AccessType;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.GenerationType;

import javax.persistence.Id;

import javax.persistence.OneToOne;

import javax.persistence.Table;

import org.hibernate.annotations.Cascade;

@Entity

@Table(name = "EMPLOYEE")

@Access(value=AccessType.FIELD)

public class Employee {

@Id

@GeneratedValue(strategy = GenerationType.IDENTITY)

@Column(name = "emp\_id")

private long id;

@Column(name = "emp\_name")

private String name;

@OneToOne(mappedBy = "employee")

@Cascade(value = org.hibernate.annotations.CascadeType.ALL)

private Address address;

//getter setter methods

}

package com.journaldev.hibernate.model;

import javax.persistence.Access;

import javax.persistence.AccessType;

import javax.persistence.Column;

import javax.persistence.Entity;

import javax.persistence.GeneratedValue;

import javax.persistence.Id;

import javax.persistence.OneToOne;

import javax.persistence.PrimaryKeyJoinColumn;

import javax.persistence.Table;

import org.hibernate.annotations.GenericGenerator;

import org.hibernate.annotations.Parameter;

@Entity

@Table(name = "ADDRESS")

@Access(value=AccessType.FIELD)

public class Address {

@Id

@Column(name = "emp\_id", unique = true, nullable = false)

@GeneratedValue(generator = "gen")

@GenericGenerator(name = "gen", strategy = "foreign", parameters = { @Parameter(name = "property", value = "employee") })

private long id;

@Column(name = "address\_line1")

private String addressLine1;

@OneToOne

@PrimaryKeyJoinColumn

private Employee employee;

//getter setter methods

}

What is Hibernate SessionFactory and how to configure it?

SessionFactory is the factory class used to get the Session objects. SessionFactory is responsible to read the hibernate configuration parameters and connect to the database and provide Session objects. Usually an application has a single SessionFactory instance and threads servicing client requests obtain Session instances from this factory.

The internal state of a SessionFactory is immutable. Once it is created this internal state is set. This internal state includes all of the metadata about Object/Relational Mapping.

SessionFactory also provide methods to get the Class metadata and Statistics instance to get the stats of query executions, second level cache details etc.

Hibernate SessionFactory is thread safe?

Internal state of SessionFactory is immutable, so it’s thread safe. Multiple threads can access it simultaneously to get Session instances.

What is Hibernate Session and how to get it?

Hibernate Session is the interface between java application layer and hibernate. This is the core interface used to perform database operations. Lifecycle of a session is bound by the beginning and end of a transaction.

Session provide methods to perform create, read, update and delete operations for a persistent object. We can execute HQL queries, SQL native queries and create criteria using Session object.

Hibernate Session is thread safe?

Hibernate Session object is not thread safe, every thread should get it’s own session instance and close it after it’s work is finished.

What is difference between openSession and getCurrentSession?

Hibernate SessionFactory getCurrentSession() method returns the session bound to the context. But for this to work, we need to configure it in hibernate configuration file. Since this session object belongs to the hibernate context, we don’t need to close it. Once the session factory is closed, this session object gets closed.

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

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

There is another method openStatelessSession() that returns stateless session, for more details with examples please read [Hibernate openSession vs getCurrentSession](http://www.journaldev.com/3522/hibernate-sessionfactory).

What is difference between Hibernate Session get() and load() method?

Hibernate session comes with different methods to load data from database. get and load are most used methods, at first look they seems similar but there are some differences between them.

get() loads the data as soon as it’s called whereas load() returns a proxy object and loads data only when it’s actually required, so load() is better because it support lazy loading.

Since load() throws exception when data is not found, we should use it only when we know data exists.

We should use get() when we want to make sure data exists in the database.

For clarification regarding the differences, please read [Hibernate get vs load](http://www.journaldev.com/3472/hibernate-session-get-vs-load-difference-with-examples).

What is hibernate caching? Explain Hibernate first level cache?

As the name suggests, hibernate caches query data to make our application faster. Hibernate Cache can be very useful in gaining fast application performance if used correctly. The idea behind cache is to reduce the number of database queries, hence reducing the throughput time of the application.

Hibernate first level cache is associated with the Session object. Hibernate first level cache is enabled by default and there is no way to disable it. However hibernate provides methods through which we can delete selected objects from the cache or clear the cache completely.  
Any object cached in a session will not be visible to other sessions and when the session is closed, all the cached objects will also be lost.

For better explanation, please read [Hibernate First Level Cache](http://www.journaldev.com/2969/hibernate-caching-first-level-cache).

How to configure Hibernate Second Level Cache using EHCache?

EHCache is the best choice for utilizing hibernate second level cache. Following steps are required to enable EHCache in hibernate application.

Add hibernate-ehcache dependency in your maven project, if it’s not maven then add corresponding jars.

<dependency>

<groupId>org.hibernate</groupId>

<artifactId>hibernate-ehcache</artifactId>

<version>4.3.5.Final</version>

</dependency>

Add below properties in hibernate configuration file.

<property name="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.EhCacheRegionFactory</property>

<!-- For singleton factory -->

<!-- <property name="hibernate.cache.region.factory\_class">org.hibernate.cache.ehcache.SingletonEhCacheRegionFactory</property>

-->

<!-- enable second level cache and query cache -->

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

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

<property name="net.sf.ehcache.configurationResourceName">/myehcache.xml</property>

Create EHCache configuration file, a sample file myehcache.xml would look like below.

<?xml version="1.0" encoding="UTF-8"?>

<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:noNamespaceSchemaLocation="ehcache.xsd" updateCheck="true"

monitoring="autodetect" dynamicConfig="true">

<diskStore path="java.io.tmpdir/ehcache" />

<defaultCache maxEntriesLocalHeap="10000" eternal="false"

timeToIdleSeconds="120" timeToLiveSeconds="120" diskSpoolBufferSizeMB="30"

maxEntriesLocalDisk="10000000" diskExpiryThreadIntervalSeconds="120"

memoryStoreEvictionPolicy="LRU" statistics="true">

<persistence strategy="localTempSwap" />

</defaultCache>

<cache name="employee" maxEntriesLocalHeap="10000" eternal="false"

timeToIdleSeconds="5" timeToLiveSeconds="10">

<persistence strategy="localTempSwap" />

</cache>

<cache name="org.hibernate.cache.internal.StandardQueryCache"

maxEntriesLocalHeap="5" eternal="false" timeToLiveSeconds="120">

<persistence strategy="localTempSwap" />

</cache>

<cache name="org.hibernate.cache.spi.UpdateTimestampsCache"

maxEntriesLocalHeap="5000" eternal="true">

<persistence strategy="localTempSwap" />

</cache>

</ehcache>

Annotate entity beans with @Cache annotation and caching strategy to use. For example,

import org.hibernate.annotations.Cache;

import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity

@Table(name = "ADDRESS")

@Cache(usage=CacheConcurrencyStrategy.READ\_ONLY, region="employee")

public class Address {

}

That’s it, we are done. Hibernate will use the EHCache for second level caching, read [Hibernate EHCache Example](http://www.journaldev.com/2980/hibernate-ehcache-hibernate-second-level-cache) for a complete example with explanation.

What are different states of an entity bean?

An entity bean instance can exist is one of the three states.

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

A New instance of  a persistent class which is not associated with a ***Session***, has no representation in the ***database***and no identifier value is considered ***transient*** by Hibernate:

1. UserDetail user = **new** UserDetail();
2. user.setUserName("Dinesh Rajput");
3. // user is in a transient state

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

A persistent instance has a representation in the ***database***, an identifier value and is associated with a ***Session***. You can make a transient instance persistent by associating it with a ***Session***:

1. Long id = (Long) session.save(user);
2. // user is now in a persistent state

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

**Detached State:**  
Now, if we close the ***Hibernate Session***, the ***persistent*** instance will become a ***detached*** instance: it isn't attached to a ***Session***anymore (but can still be modified and reattached to a new Session later though).

1. session.close();
2. //user in detached state

**Difference between Transient and Detached States:**  
 Transient objects do not have association with the databases and session objects. They are simple objects and not persisted to the database. Once the last reference is lost, that means the object itself is lost. And of course , garbage collected. The commits and rollbacks will have no effects on these objects. They can become into persistent objects through the save method calls of Session object.  
 The detached object has corresponding entries in the database. These are persistent and not connected to the Session object. These objects have the synchronized data with the database when the session was closed. Since then, the change may be done in the database which makes this object stale. The detached object can be reattached after certain time to another object in order to become persistent again.

What is use of Hibernate Session merge() call?

Hibernate merge can be used to update existing values, however this method create a copy from the passed entity object and return it. The returned object is part of persistent context and tracked for any changes, passed object is not tracked. For example program, read [Hibernate merge](http://www.journaldev.com/3481/hibernate-session-merge-vs-update-save-saveorupdate-persist-example).

What is difference between Hibernate save(), saveOrUpdate() and persist() methods?

Hibernate save can be used to save entity to database. Problem with save() is that it can be invoked without a transaction and if we have mapping entities, then only the primary object gets saved causing data inconsistencies. Also save returns the generated id immediately.

Hibernate persist is similar to save with transaction. I feel it’s better than save because we can’t use it outside the boundary of transaction, so all the object mappings are preserved. Also persist doesn’t return the generated id immediately, so data persistence happens when needed.

Hibernate saveOrUpdate results into insert or update queries based on the provided data. If the data is present in the database, update query is executed. We can use saveOrUpdate() without transaction also, but again you will face the issues with mapped objects not getting saved if session is not flushed. For example usage of these methods, read [Hibernate save vs persist](http://www.journaldev.com/3481/hibernate-session-merge-vs-update-save-saveorupdate-persist-example).

What will happen if we don’t have no-args constructor in Entity bean?

Hibernate uses [Reflection API](http://www.journaldev.com/1789/java-reflection-example-tutorial) to create instance of Entity beans, usually when you call get() or load() methods. The method Class.newInstance() is used for this and it requires no-args constructor. So if you won’t have no-args constructor in entity beans, hibernate will fail to instantiate it and you will get HibernateException.

What is difference between sorted collection and ordered collection, which one is better?

When we use Collection API sorting algorithms to sort a collection, it’s called sorted list. For small collections, it’s not much of an overhead but for larger collections it can lead to slow performance and OutOfMemory errors. Also the entity beans should implement Comparable or Comparator interface for it to work, read more at [java object list sorting](http://www.journaldev.com/780/comparable-and-comparator-in-java-example).

If we are using Hibernate framework to load collection data from database, we can use it’s Criteria API to use “order by” clause to get ordered list. Below code snippet shows you how to get it.

List<Employee> empList = session.createCriteria(Employee.class)

.addOrder(Order.desc("id")).list();

Ordered list is better than sorted list because the actual sorting is done at database level, that is fast and doesn’t cause memory issues.

What are the collection types in Hibernate?

There are five collection types in hibernate used for one-to-many relationship mappings.

Bag

Set

List

Array

Map

How to implement Joins in Hibernate?

There are various ways to implement joins in hibernate.

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

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

We can fire native sql query and use join keyword.

Why we should not make Entity Class final?

Hibernate use proxy classes for lazy loading of data, only when it’s needed. This is done by extending the entity bean, if the entity bean will be final then lazy loading will not be possible, hence low performance.

What is HQL and what are it’s benefits?

Hibernate Framework comes with a powerful object-oriented query language – Hibernate Query Language (HQL). It’s very similar to SQL except that we use Objects instead of table names, that makes it more close to object oriented programming.

Hibernate query language is case-insensitive except for java class and variable names. So SeLeCT is the same as sELEct is the same as SELECT, but com.journaldev.model.Employee is not same as com.journaldev.model.EMPLOYEE.

The HQL queries are cached but we should avoid it as much as possible, otherwise we will have to take care of associations. However it’s a better choice than native sql query because of Object-Oriented approach. Read more at [HQL Example](http://www.journaldev.com/2954/hibernate-query-language-hql-example-tutorial).

What is Query Cache in Hibernate?

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

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

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

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

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

query.setCacheable(true);

query.setCacheRegion("ALL\_EMP");

Can we execute native sql query in hibernate?

Hibernate provide option to execute native SQL queries through the use of SQLQuery object.

For normal scenarios, it is however not the recommended approach because we loose benefits related to hibernate association and hibernate first level caching. Read more at [Hibernate Native SQL Query Example](http://www.journaldev.com/3422/hibernate-native-sql-query-example).

What is the benefit of native sql query support in hibernate?

Native SQL Query comes handy when we want to execute database specific queries that are not supported by Hibernate API such as query hints or the CONNECT keyword in Oracle Database.

What is Named SQL Query?

Hibernate provides Named Query that we can define at a central location and use them anywhere in the code. We can created named queries for both HQL and Native SQL.

Hibernate Named Queries can be defined in Hibernate mapping files or through the use of JPA annotations @NamedQuery and @NamedNativeQuery.

What are the benefits of Named SQL Query?

Hibernate Named Query helps us in grouping queries at a central location rather than letting them scattered all over the code.  
Hibernate Named Query syntax is checked when the hibernate session factory is created, thus making the application fail fast in case of any error in the named queries.  
Hibernate Named Query is global, means once defined it can be used throughout the application.

However one of the major disadvantage of Named query is that it’s hard to debug, because we need to find out the location where it’s defined.

What is the benefit of Hibernate Criteria API?

Hibernate provides Criteria API that is more object oriented for querying the database and getting results. We can’t use Criteria to run update or delete queries or any DDL statements. It’s only used to fetch the results from the database using more object oriented approach.

Some of the common usage of Criteria API are:

Criteria API provides Projection that we can use for aggregate functions such as sum(), min(), max() etc.

Criteria API can be used with ProjectionList to fetch selected columns only.

Criteria API can be used for join queries by joining multiple tables, useful methods are createAlias(), setFetchMode() and setProjection()

Criteria API can be used for fetching results with conditions, useful methods are add() where we can add Restrictions.

Criteria API provides addOrder() method that we can use for ordering the results.

Learn some quick examples at [Hibernate Criteria Example](http://www.journaldev.com/2963/hibernate-criteria-example-tutorial).

How to log hibernate generated sql queries in log files?

We can set below property for hibernate configuration to log SQL queries.

<property name="hibernate.show\_sql">true</property>

However we should use it only in Development or Testing environment and turn it off in production environment.

What is Hibernate Proxy and how it helps in lazy loading?

Hibernate uses proxy object to support lazy loading. Basically when you load data from tables, hibernate doesn’t load all the mapped objects. As soon as you reference a child or lookup object via getter methods, if the linked entity is not in the session cache, then the proxy code will go to the database and load the linked object. It uses javassist to effectively and dynamically generate sub-classed implementations of your entity objects.

How to implement relationships in hibernate?

We can easily implement one-to-one, one-to-many and many-to-many relationships in hibernate. It can be done using JPA annotations as well as XML based configurations. For better understanding, you should go through following tutorials.

[Hibernate One to One Mapping](http://www.journaldev.com/2916/hibernate-one-to-one-mapping-example-annotation)

[Hibernate One to Many Mapping](http://www.journaldev.com/2924/hibernate-one-to-many-mapping-annotation)

[Hibernate Many to Many Mapping](http://www.journaldev.com/2934/hibernate-many-to-many-mapping-join-tables)

How transaction management works in Hibernate?

Transaction management is very easy in hibernate because most of the operations are not permitted outside of a transaction. So after getting the session from SessionFactory, we can call session beginTransaction() to start the transaction. This method returns the Transaction reference that we can use later on to either commit or rollback the transaction.

Overall hibernate transaction management is better than JDBC transaction management because we don’t need to rely on exceptions for rollback. Any exception thrown by session methods automatically rollback the transaction.

What is cascading and what are different types of cascading?

When we have relationship between entities, then we need to define how the different operations will affect the other entity. This is done by cascading and there are different types of it.

Here is a simple example of applying cascading between primary and secondary entities.

import org.hibernate.annotations.Cascade;

@Entity

@Table(name = "EMPLOYEE")

public class Employee {

@OneToOne(mappedBy = "employee")

@Cascade(value = org.hibernate.annotations.CascadeType.ALL)

private Address address;

}

Note that Hibernate CascadeType enum constants are little bit different from JPA javax.persistence.CascadeType, so we need to use the Hibernate CascadeType and Cascade annotations for mappings, as shown in above example.  
Commonly used cascading types as defined in CascadeType enum are:

None: No Cascading, it’s not a type but when we don’t define any cascading then no operations in parent affects the child.

ALL: Cascades save, delete, update, evict, lock, replicate, merge, persist. Basically everything

SAVE\_UPDATE: Cascades save and update, available only in hibernate.

DELETE: Corresponds to the Hibernate native DELETE action, only in hibernate.

DETATCH, MERGE, PERSIST, REFRESH and REMOVE – for similar operations

LOCK: Corresponds to the Hibernate native LOCK action.

REPLICATE: Corresponds to the Hibernate native REPLICATE action.

How to integrate log4j logging in hibernate application?

Hibernate 4 uses JBoss logging rather than slf4j used in earlier versions. For log4j configuration, we need to follow below steps.

Add log4j dependencies for maven project, if not maven then add corresponding jar files.

Create log4j.xml configuration file or log4j.properties file and keep it in the classpath. You can keep file name whatever you want because we will load it in next step.

For standalone projects, use static block to configure log4j using DOMConfigurator or PropertyConfigurator. For web applications, you can use ServletContextListener to configure it.

That’s it, our setup is ready. Create org.apache.log4j.Logger instance in the java classes and start logging. For complete example code, you should go through [Hibernate log4j example](http://www.journaldev.com/2984/hibernate-log4j-logging) and [Servlet log4j example](http://www.journaldev.com/1997/servlet-jdbc-database-connection-example).

How to use application server JNDI DataSource with Hibernate framework?

For web applications, it’s always best to allow servlet container to manage the connection pool. That’s why we define JNDI resource for DataSource and we can use it in the web application. It’s very easy to use in Hibernate, all we need is to remove all the database specific properties and use below property to provide the JNDI DataSource name.

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

For a complete example, go through [Hibernate JNDI DataSource Example](http://www.journaldev.com/2905/hibernate-tomcat-jndi-datasource-example-tutorial).

How to integrate Hibernate and Spring frameworks?

Spring is one of the most used Java EE Framework and Hibernate is the most popular ORM framework. That’s why Spring Hibernate combination is used a lot in enterprise applications. The best part with using Spring is that it provides out-of-box integration support for Hibernate with **Spring ORM** module. Following steps are required to integrate Spring and Hibernate frameworks together.

Add hibernate-entitymanager, hibernate-core and spring-orm dependencies.

Create Model classes and corresponding DAO implementations for database operations. Note that DAO classes will use SessionFactory that will be injected by Spring Bean configuration.

If you are using Hibernate 3, you need to configure org.springframework.orm.hibernate3.LocalSessionFactoryBean or org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBeanin Spring Bean configuration file. For Hibernate 4, there is single class org.springframework.orm.hibernate4.LocalSessionFactoryBean that should be configured.

Note that we don’t need to use Hibernate Transaction Management, we can leave it to Spring declarative transaction management using @Transactional annotation.

For complete example go through [Spring Hibernate Integration](http://www.journaldev.com/3524/spring-hibernate-integration-example-tutorial) and [Spring MVC Hibernate Integration](http://www.journaldev.com/3531/spring-mvc-hibernate-mysql-integration-crud-example-tutorial).

What is HibernateTemplate class?

When Spring and Hibernate integration started, Spring ORM provided two helper classes – HibernateDaoSupport and HibernateTemplate. The reason to use them was to get the Session from Hibernate and get the benefit of Spring transaction management. However from Hibernate 3.0.1, we can use SessionFactory getCurrentSession() method to get the current session and use it to get the spring transaction management benefits. If you go through above examples, you will see how easy it is and that’s why we should not use these classes anymore.

One other benefit of HibernateTemplate was exception translation but that can be achieved easily by using @Repository annotation with service classes, shown in above spring mvc example. This is a trick question to judge your knowledge and whether you are aware of recent developments or not.

How to integrate Hibernate with Servlet or Struts2 web applications?

Hibernate integration with Servlet or Struts2 needs to be done using ServletContextListener, a complete example can be found at [Hibernate Struts2 Integration Example](http://www.journaldev.com/3557/struts2-hibernate-integration-example-tutorial).

Which design patterns are used in Hibernate framework?

Some of the design patterns used in Hibernate Framework are:

Domain Model Pattern – An object model of the domain that incorporates both behavior and data.

Data Mapper – A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.

[Proxy Pattern](http://www.journaldev.com/1572/proxy-design-pattern) for lazy loading

[Factory pattern](http://www.journaldev.com/1392/factory-design-pattern-in-java) in SessionFactory

What are best practices to follow with Hibernate framework?

Some of the best practices to follow in Hibernate are:

Always check the primary key field access, if it’s generated at the database layer then you should not have a setter for this.

By default hibernate set the field values directly, without using setters. So if you want hibernate to use setters, then make sure proper access is defined as @Access(value=AccessType.PROPERTY).

If access type is property, make sure annotations are used with getter methods and not setter methods. Avoid mixing of using annotations on both filed and getter methods.

Use native sql query only when it can’t be done using HQL, such as using database specific feature.

If you have to sort the collection, use ordered list rather than sorting it using Collection API.

Use named queries wisely, keep it at a single place for easy debugging. Use them for commonly used queries only. For entity specific query, you can keep them in the entity bean itself.

For web applications, always try to use JNDI DataSource rather than configuring to create connection in hibernate.

Avoid Many-to-Many relationships, it can be easily implemented using bidirectional One-to-Many and Many-to-One relationships.

For collections, try to use Lists, maps and sets. Avoid array because you don’t get benefit of lazy loading.

Do not treat exceptions as recoverable, roll back the Transaction and close the Session. If you do not do this, Hibernate cannot guarantee that in-memory state accurately represents the persistent state.

Prefer DAO pattern for exposing the different methods that can be used with entity bean

Prefer lazy fetching for associations

What is Hibernate Validator Framework?

Data validation is integral part of any application. You will find data validation at presentation layer with the use of Javascript, then at the server side code before processing it. Also data validation occurs before persisting it, to make sure it follows the correct format.

Validation is a cross cutting task, so we should try to keep it apart from our business logic. That’s why JSR303 and JSR349 provides specification for validating a bean by using annotations. Hibernate Validator provides the reference implementation of both these bean validation specs. Read more at [Hibernate Validation Example](http://www.journaldev.com/3626/hibernate-validator-jsr303-example-tutorial).

What is the benefit of Hibernate Tools Eclipse plugin?

Hibernate Tools plugin helps us in writing hibernate configuration and mapping files easily. The major benefit is the content assist to help us with properties or xml tags to use. It also validates them against the Hibernate DTD files, so we know any mistakes before hand. Learn how to install and use at [Hibernate Tools Eclipse Plugin](http://www.journaldev.com/2940/hibernate-tools-eclipse-plugin).

1. What’s Hibernate?

Hibernate is a popular framework of Java which allows an efficient Object Relational mapping using configuration files in XML format. After java objects mapping to database tables, database is used and handled using Java objects without writing complex database queries.

2. What is ORM?  
ORM (Object Relational Mapping) is the fundamental concept of Hibernate framework which maps database tables with Java Objects and then provides various API’s to perform different types of operations on the data tables.

3. How properties of a class are mapped to the columns of a database table in Hibernate?

Mappings between class properties and table columns are specified in XML file as in the below example:

4. What’s the usage of Configuration Interface in hibernate?

Configuration interface of hibernate framework is used to configure hibernate. It’s also used to bootstrap hibernate. Mapping documents of hibernate are located using this interface.

5. How can we use new custom interfaces to enhance functionality of built-in interfaces of hibernate?

We can use extension interfaces in order to add any required functionality which isn’t supported by built-in interfaces.

6. Should all the mapping files of hibernate have .hbm.xml extension to work properly?

No, having .hbm.xml extension is a convention and not a requirement for hibernate mapping file names. We can have any extension for these mapping files.

7. How do we create session factory in hibernate?

To create a session factory in hibernate, an object of configuration is created first which refers to the path of configuration file and then for that configuration, session factory is created as given in the example below:

|  |  |
| --- | --- |
| 1  2  3  4 | Configuration config = new Configuration();  config.addResource(&amp;amp;quot;myinstance/configuration.hbm.xml&amp;amp;quot;);  config.setProperties( System.getProperties() );  SessionFactory sessions = config.buildSessionFactory(); |

8. What are POJOs and what’s their significance?

POJOs( Plain Old Java Objects) are java beans with proper getter and setter methods for each and every properties.  
Use of POJOs instead of simple java classes results in an efficient and well constructed code.

9. What’s HQL?  
HQL is the query language used in Hibernate which is an extension of SQL. HQL is very efficient, simple and flexible query language to do various type of operations on relational database without writing complex database queries.

10. How can we invoke stored procedures in hibernate?  
In hibernate we can execute stored procedures using code as below:

[xml]

<sql-query name=”getStudents” callable=”true”>  
<return alias=”st” class=”Student”>  
<return-property name=”std\_id” column=”STD\_ID”/>  
<return-property name=”s\_name” column=”STD\_NAME”/>  
<return-property name=”s\_dept” column=”STD\_DEPARTMENT”/>  
{ ? = call selectStudents() }  
</return>  
</sql-query>

[/xml]

11. What is criteria API?

Criteria is a simple yet powerful API of hibernate which is used to retrieve entities through criteria object composition.

12. What are the benefits of using Hibernate template?  
Following are some key benefits of using Hibernate template:  
a. Session closing is automated.  
b. Interaction with hibernate session is simplified.  
c. Exception handling is automated.

13. How can we see hibernate generated SQL on console?  
We need to add following in hibernate configuration file to enable viewing SQL on the console for debugging purposes:

[xml]

<property name=”show\_sql”>true</property>

[/xml]

14. What are the two types of collections in hibernate?  
Following are the two types of collections in hibernate:  
a. Sorted Collection  
b. Order Collection

15. What’s the difference between session.save() and session.saveOrUpdate() methods in hibernate?  
Sessionsave() method saves a record only if it’s unique with respect to its primary key and will fail to insert if primary key already exists in the table.  
saveOrUpdate() method inserts a new record if primary key is unique and will update an existing record if primary key exists in the table already.

16. What the benefits are of hibernate over JDBC?  
a. Hibernate can be used seamlessly with any type of database as its database independent while in case of JDBC, developer has to write database specific queries.  
b. Using hibernate, developer doesn’t need to be an expert of writing complex queries as HQL simplifies query writing process while in case of JDBC, its job of developer to write and tune queries.  
c. In case of hibernate, there is no need to create connection pools as hibernate does all connection handling automatically while in case of JDBC, connection pools need to be created.

17. How can we get hibernate statistics?  
We can get hibernate statistics using getStatistics() method of SessionFactory class as shown below:  
SessionFactory.getStatistics()

18. What is transient instance state in Hibernate?  
If an instance is not associated with any persistent context and also, it has never been associated with any persistent context, then it’s said to be in transient state.

19. How can we reduce database write action times in Hibernate?  
Hibernate provides dirty checking feature which can be used to reduce database write times. Dirty checking feature of hibernate updates only those fields which require a change while keeps others unchanged.

20. What’s the usage of callback interfaces in hibernate?  
Callback interfaces of hibernate are useful in receiving event notifications from objects. For example, when an object is loaded or deleted, an event is generated and notification is sent using callback interfaces.

21. When an instance goes in detached state in hibernate?  
When an instance was earlier associated with some persistent context (e.g. a table) and is no longer associated, it’s called to be in detached state.

22. What the four ORM levels are in hibernate?  
Following are the four ORM levels in hibernate:  
a. Pure Relational  
b. Light Object Mapping  
c. Medium Object Mapping  
d. Full Object Mapping

23. What’s transaction management in hibernate? How it works?  
Transaction management is the process of managing a set of statements or commands. In hibernate; transaction management is done by transaction interface as shown in below code:

[java]  
Session s = null;  
Transaction tr = null;  
try {  
s = sessionFactory.openSession();  
tr = s.beginTransaction();  
doTheAction(s);  
tr.commit();  
} catch (RuntimeException exc) {  
tr.rollback();  
} finally {  
s.close();  
}

[/java]

24. What the two methods are of hibernate configuration?  
We can use any of the following two methods of hibernate configuration:  
a. XML based configuration ( using hibernate.cfg.xml file)  
b. Programmatic configuration ( Using code logic)

25. What is the default cache service of hibernate?  
Hibernate supports multiple cache services like EHCache, OSCache, SWARMCache and TreeCache and default cache service of hibernate is EHCache.

26. What are the two mapping associations used in hibernate?  
In hibernate; we have following two types of mapping associations between entities:  
a. One-to-One Association  
b. Many-to-Many Association

27. What’s the usage of Hibernate QBC API?  
Hibernate Query By Criteria (QBC) API is used to create queries by manipulation of criteria objects at runtime.  
28. In how many ways, objects can be fetched from database in hibernate?  
Hibernate provides following four ways to fetch objects from database:  
a. Using HQL  
b. Using identifier  
c. Using Criteria API  
d. Using Standard SQL

29. How primary key is created by using hibernate?  
Database primary key is specified in the configuration file hbm.xml. Generator can also be used to specify how primary key is being created in the database.  
In the below example, deptId acts as primary key:

[xml]  
<id name=”deptId” type=”string” >  
<column name=”columnId” length=”30″/>  
<generator/>  
</id>  
[/xml]

30. How can we reattach any detached objects in Hibernate?

Objects which have been detached and are no longer associated with any persistent entities can be reattached by calling session.merge() method of session class.  
31. What are different ways to disable hibernate second level cache?

Hibernate second level cache can be disabled using any of the following ways:  
a. By setting use\_second\_level\_cache as false.  
b. By using CACHEMODE.IGNORE  
c. Using cache provider as org.hibernate.cache.NoCacheProvider

32. What is ORM metadata?  
All the mapping between classes and tables, properties and columns, Java types and SQL types etc is defined in ORM metadata.

33. Which one is the default transaction factory in hibernate?  
With hibernate 3.2, default transaction factory is JDBCTransactionFactory.

34. What’s the role of JMX in hibernate?  
Java Applications and components are managed in hibernate by a standard API called JMX API. JMX provides tools for development of efficient and robust distributed, web based solutions.  
35. How can we bind hibernate session factory to JNDI ?  
Hibernate session factory can be bound to JNDI by making configuration changes in hibernate.cfg file.

36. In how many ways objects can be identified in Hibernate?  
Object identification can be done in hibernate in following three ways:  
a. Using Object Identity: Using == operator.  
b. Using Object Equality: Using equals() method.  
c. Using database identity: Relational database objects can be identified if they represent same row.

37. What different fetching strategies are of hibernate?  
Following fetching strategies are available in hibernate:  
1. Join Fetching  
2. Batch Fetching  
3. Select Fetching  
4. Sub-select Fetching  
38. How mapping of java objects is done with database tables?  
To map java objects with database tables, we need to have Java beans properties names same as column names of a database table. Then mapping is provided in hbm.xml file as given below:

[xml]  
<hibernate-mapping>  
<class name=”Student”  table=”tbl\_student”>  
<property  column=”studentname” length=”255″  
name=”studentName” not-null=”true”  type=”java.lang.String”/>  
<property  column=”studentDisciplne” length=”255″  
name=”studentDiscipline” not-null=”true”  type=”java.lang.String”/>  
</class>  
</hibernate-mapping>  
[/xml]

39. What are derived properties in hibernate?  
Derived properties are those properties which are not mapped to any columns of a database table. Such properties are calculated at runtime by evaluation of any expressions.

40. What is meant by a Named SQL Query in hibernate and how it’s used?  
Named SQL queries are those queries which are defined in mapping file and are called as required anywhere.  
For example, we can write a SQL query in our XML mapping file as follows:

[xml]

<sql-query name = “studentdetails”>  
<return alias=”std”/>  
SELECT std.STUDENT\_ID AS {std.STUDENT\_ID},  
std.STUDENT\_DISCIPLINE AS {std.discipline},

FROM Student std WHERE std.NAME LIKE :name  
</sql-query>

[/xml]

Then this query can be called as follows:

[java]

List students = session.getNamedQuery(&amp;quot;studentdetails&amp;quot;)  
.setString(&amp;quot;TomBrady&amp;quot;, name)  
.setMaxResults(50)  
.list();

[/java]

41. What’s the difference between load() and get() method in hibernate?  
Load() methods results in an exception if the required records isn’t found in the database while get() method returns null when records against the id isn’t found in the database.  
So, ideally we should use Load() method only when we are sure about existence of records against an id.

42. What’s the use of version property in hibernate?  
Version property is used in hibernate to know whether an object is in transient state or in detached state.

43. What is attribute oriented programming?  
In Attribute oriented programming, a developer can add Meta data (attributes) in the java source code to add more significance in the code. For Java (hibernate), attribute oriented programming is enabled by an engine called XDoclet.

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

45. Does hibernate support polymorphism?  
Yes, hibernate fully supports polymorphism. Polymorphism queries and polymorphism associations are supported in all mapping strategies of hibernate.

46. What the three inheritance models are of hibernate?  
Hibernate has following three inheritance models:  
a. Tables Per Concrete Class  
b. Table per class hierarchy  
c. Table per sub-class

47. How can we map the classes as immutable?  
If we don’t want an application to update or delete objects of a class in hibernate, we can make the class as immutable by setting mutable=false

48. What’s general hibernate flow using RDBMS?  
General hibernate flow involving RDBMS is as follows:  
a. Load configuration file and create object of configuration class.  
b. Using configuration object, create sessionFactory object.  
c. From sessionFactory, get one session.  
d. Create HQL query.  
e. Execute HQL query and get the results. Results will be in the form of a list.

49. What is Light Object Mapping?  
Light Object Mapping is one of the levels of ORM quality in which all entities are represented as classes and they are mapped manually.

50. What’s difference between managed associations and hibernate associations?  
Managed associations relate to container management persistence and are bi-directional while hibernate associations are unidirectional.