1) What is hibernate?

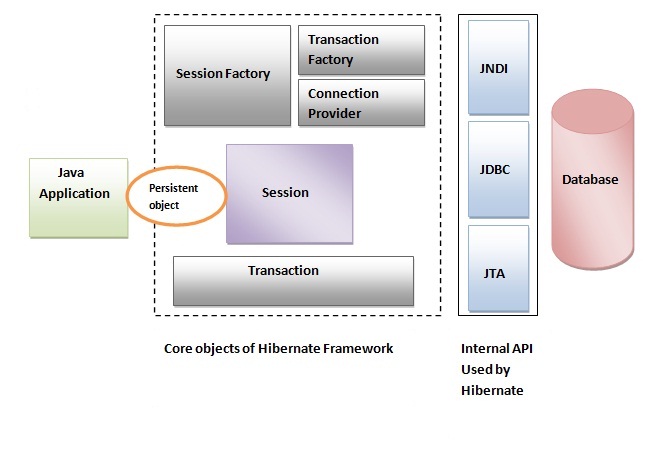
Hibernate is an open-source and lightweight ORM tool that is used to store, manipulate and retrieve data from the database.

2) What is ORM?

ORM is an acronym for Object/Relational mapping. It is a programming strategy to map object with the data stored in the database. It simplifies data creation, data manipulation and data access.

3) Explain hibernate architecture?

Hibernate architecture comprises of many interfaces such as Configuration, SessionFactory, Session, Transaction etc.



4) What are the core interfaces of Hibernate?

The core interfaces of Hibernate framework are:

1. **Session Interface –**This is the primary interface used by hibernate applications. The instances of this interface are lightweight and are inexpensive to create and destroy. Hibernate sessions are not thread safe.
2. **SessionFactory Interface –**This is a factory that delivers the session objects to hibernate application. Generally there will be a single SessionFactory for the whole application and it will be shared among all the application threads.
3. **Configuration Interface –**This interface is used to configure and bootstrap hibernate. The instance of this interface is used by the application in order to specify the location of hibernate specific mapping documents.
4. **Transaction Interface –**This is an optional interface but the above three interfaces are mandatory in each and every application. This interface abstracts the code from any kind of transaction implementations such as JDBC transaction, JTA transaction.
5. **Query and Criteria Interface –**This interface allows the user to perform queries and also control the flow of the query execution.

5) What is SessionFactory?

SessionFactory provides the instance of Session. It is a factory of Session. It holds the data of second level cache that is not enabled by default.

6) Is SessionFactory a thread-safe object?

Yes, SessionFactory is a thread-safe object, many threads cannot access it simultaneously.

7) What is Session?

It maintains a connection between hibernate application and database.

It provides methods to store, update, delete or fetch data from the database such as persist(), update(), delete(), load(), get() etc.

It is a factory of Query, Criteria and Transaction i.e. it provides factory methods to return these instances.

8) Is Session a thread-safe object?

No, Session is not a thread-safe object, many threads can access it simultaneously. In other words, you can share it between threads.

9) What is the difference between session.save() and session.persist() method?

|  |  |  |
| --- | --- | --- |
| **No.** | **save()** | **persist()** |
| 1) | returns the identifier (Serializable) of the instance. | return nothing because its return type is void. |
| 2) | Syn: public Serializable save(Object o) | Syn: public void persist(Object o) |

10) What is the difference between get and load method?

The differences between get() and load() methods are given below.

|  |  |  |
| --- | --- | --- |
| **No.** | **get()** | **load()** |
| 1) | Returns **null** if object is not found. | Throws **ObjectNotFoundException** if object is not found. |
| 2) | get() method always **hit the database**. | load() method **doesn't hit** the database. |
| 3) | It returns real object **not proxy**. | It returns **proxy object.** |
| 4) | It should be used if **you are not sure** about the existence of instance. | It should be used if **you are sure** that instance exists. |

It’s about performance

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()**.

1. session.get()

For example, in a Stock application , Stock and StockTransactions should have a “one-to-many” relationship, when you want to save a stock transaction, it’s common to declared something like below

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

StockTransactionstockTransactions=**new**StockTransaction();

*//set stockTransactions detail*

stockTransactions.setStock(stock);

session.save(stockTransactions);

Output

Hibernate:

**select** ... from mkyong.stock stock0\_

where stock0\_.STOCK\_ID=?

Hibernate:

insert into mkyong.stock\_transaction**(**...**)**

values**(**?, ?, ?, ?, ?, ?**)**

In session.get(), Hibernate will hit the database to retrieve the Stock object and put it as a reference to StockTransaction. However, this save process is extremely high demand, there may be thousand or million transactions per hour, do you think is this necessary to hit the database to retrieve the Stock object everything save a stock transaction record? After all you just need the Stock’s Id as a reference to StockTransaction.

2. 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));

StockTransactionstockTransactions=**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, throwObjectNotFoundException*

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.

Conclusion

There are no always correct solution, you have to understand the differential in between, and decide which method is best fix in your application.

11) What is the difference between update and merge method?

The differences between update() and merge() methods are given below.

|  |  |  |
| --- | --- | --- |
| **No.** | **update() method** | **merge() method** |
| 1) | Update means to edit something. | Merge means to combine something. |
| 2) | update() should be used if session doesn't contain an already persistent state with same id. It means update should be used inside the session only. After closing the session it will throw error. | merge() should be used if you don't know the state of the session, means you want to make modification at any time. |

Let's try to understand the difference by the example given below:

1. ...
2. SessionFactory factory = cfg.buildSessionFactory();
3. Session session1 = factory.openSession();
5. Employee e1 = (Employee) session1.get(Employee.**class**, Integer.valueOf(101));//passing id of employee
6. session1.close();
8. e1.setSalary(70000);
10. Session session2 = factory.openSession();
11. Employee e2 = (Employee) session1.get(Employee.**class**, Integer.valueOf(101));//passing same id
13. Transaction tx=session2.beginTransaction();
14. session2.merge(e1);
16. tx.commit();
17. session2.close();

After closing session1, e1 is in detached state. It will not be in session1 cache. So if you call update() method, it will throw an error.

Then, we opened another session and loaded the same Employee instance. If we call merge in session2, changes of e1 will be merged in e2.

12) What are the states of object in hibernate?

There are 3 states of object (instance) in hibernate.

1. **Transient**: The object is in transient state if it is just created but has no primary key (identifier) and not associated with session.
2. **Persistent**: The object is in persistent state if session is open, and you just saved the instance in the database or retrieved the instance from the database.
3. **Detached**: The object is in detached state if session is closed. After detached state, object comes to persistent state if you call lock() or update() method.

14) How to make a immutable class in hibernate?

If you mark a class as mutable="false", class will be treated as an immutable class. By default, it is mutable="true".

15) What is automatic dirty checking in hibernate?

The automatic dirty checking feature of hibernate, calls update statement automatically on the objects that are modified in a transaction.

Let's understand it by the example given below:

1. ...
2. SessionFactory factory = cfg.buildSessionFactory();
3. Session session1 = factory.openSession();
4. Transaction tx=session2.beginTransaction();
6. Employee e1 = (Employee) session1.get(Employee.**class**, Integer.valueOf(101));
8. e1.setSalary(70000);
10. tx.commit();
11. session1.close();

Here, after getting employee instance e1 and we are changing the state of e1.

After changing the state, we are committing the transaction. In such case, state will be updated automatically. This is known as dirty checking in hibernate.

16) How many types of association mapping are possible in hibernate?

There can be 4 types of association mapping in hibernate.

1. One to One
2. One to Many
3. Many to One
4. Many to Many

17) Is it possible to perform collection mapping with One-to-One and Many-to-One?

No, collection mapping can only be performed with One-to-Many and Many-to-Many

18) What is lazy loading in hibernate?

Lazy loading in hibernate improves the performance. It loads the child objects on demand.

Since Hibernate 3, lazy loading is enabled by default, you don't need to do lazy="true". It means not to load the child objects when parent is loaded.

19) What is HQL (Hibernate Query Language)?

Hibernate Query Language is known as an object oriented query language. It is like structured query language (SQL).

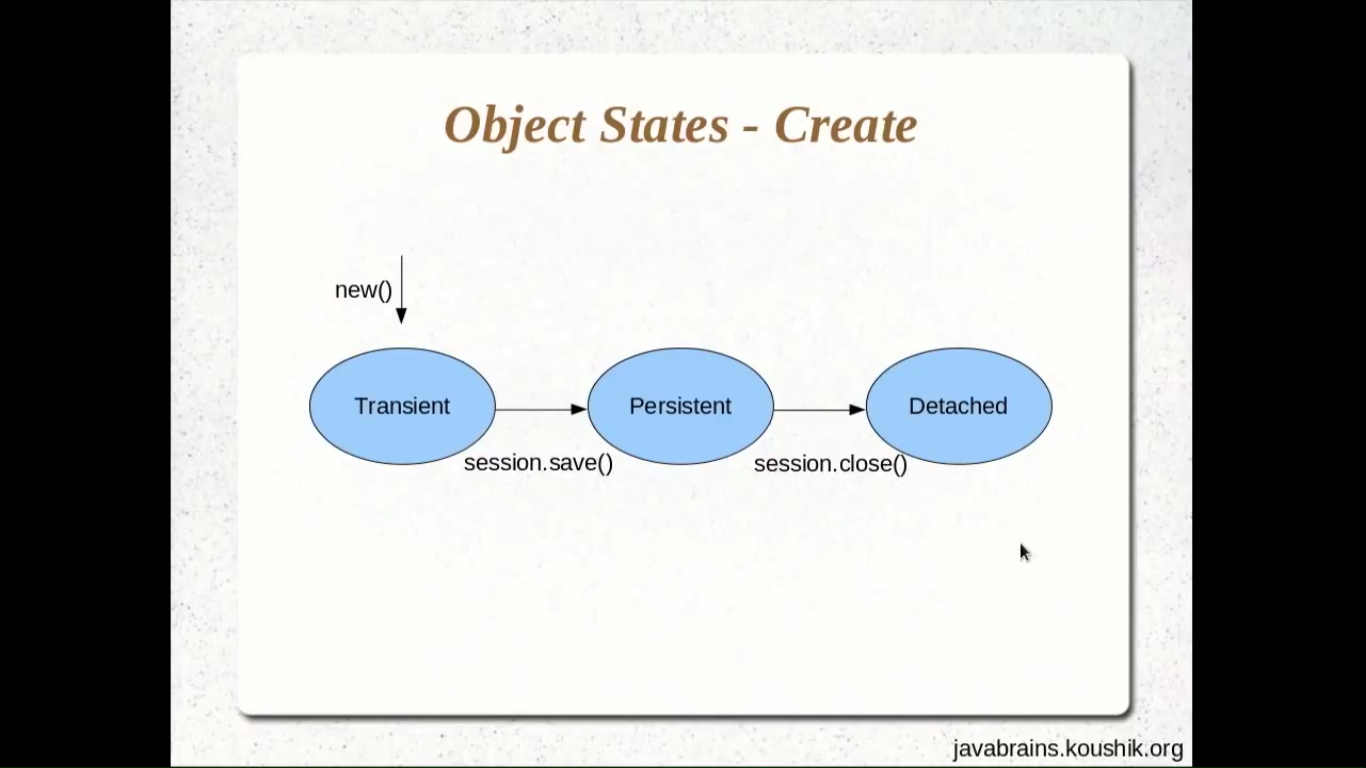
The main advantage of HQL over SQL is:

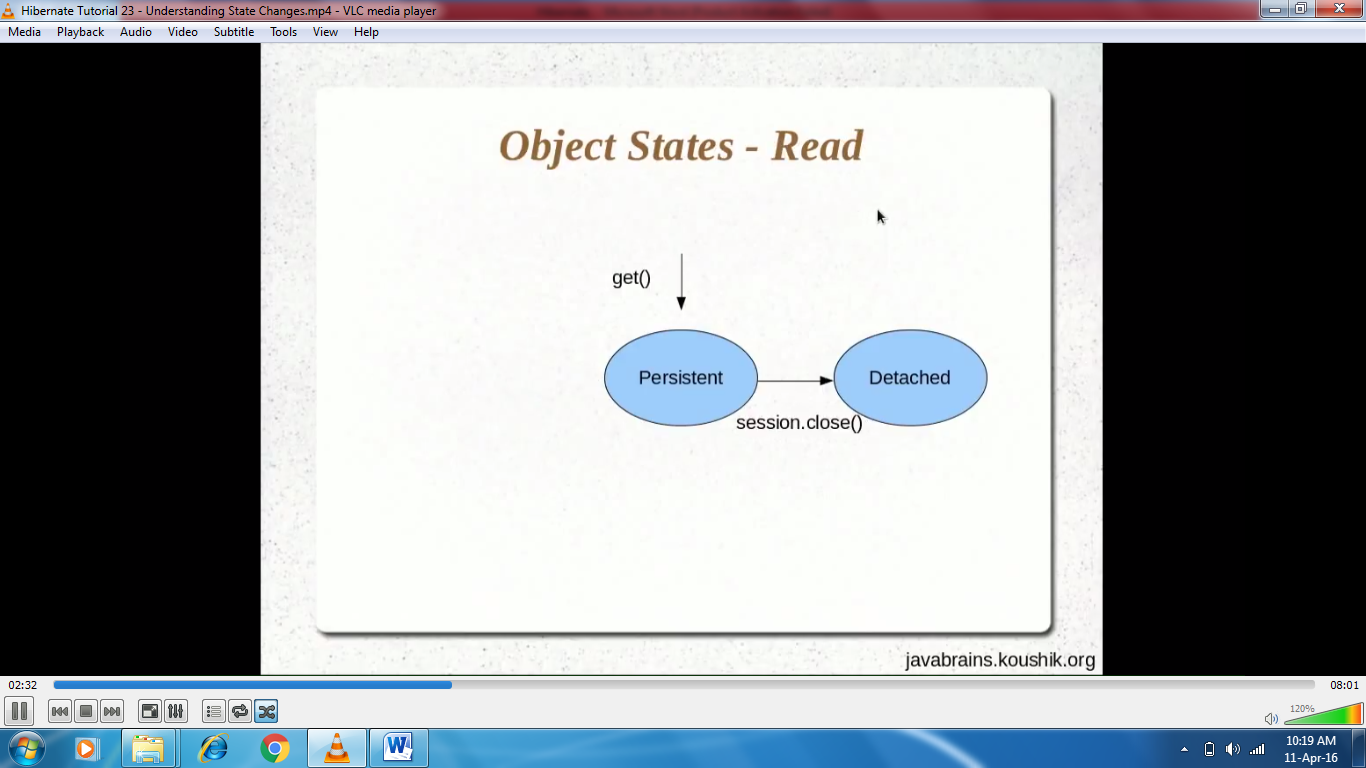
1. You don't need to learn SQL
2. Database independent
3. Simple to write query

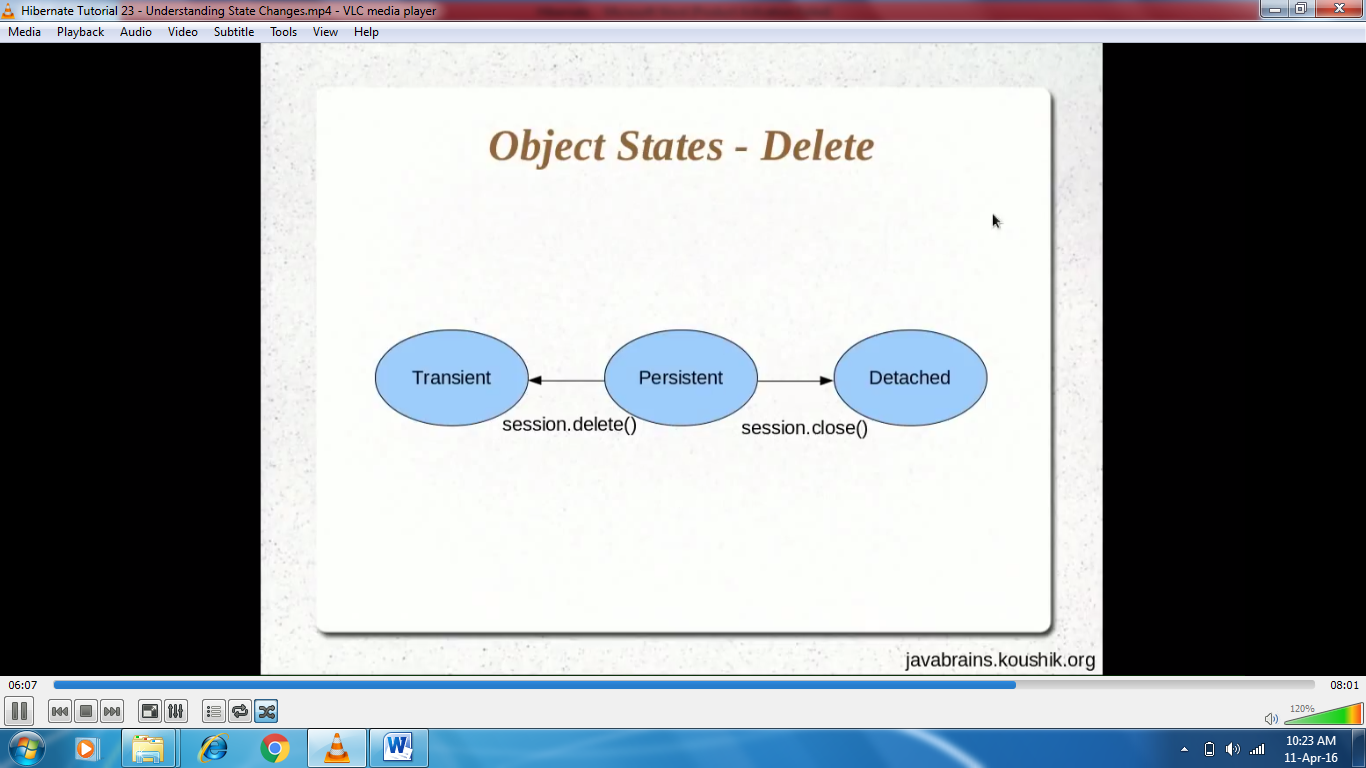
20) What is the difference between first level cache and second level cache?

|  |  |  |
| --- | --- | --- |
| **No.** | **First Level Cache** | **Second Level Cache** |
| 1) | First Level Cache is **associated with Session**. | Second Level Cache is associated with **SessionFactory**. |
| 2) | It is **enabled** by default. | It is **not enabled** by default. |









# Hibernate Inheritance Mapping:

# We can map the inheritance hierarchy classes with the table of the database. There are three inheritance mapping strategies defined in the hibernate:

1. Table Per Hierarchy(Single table strategy)
2. Table Per Concrete class(Table per class strategy)
3. Table Per Subclass(joined strategy)

#### Table Per Hierarchy

In table per hierarchy mapping, single table is required to map the whole hierarchy, an extra column (known as discriminator column) is added to identify the class. But nullable values are stored in the table .

[Table Per Hierarchy using xml file](http://www.javatpoint.com/hibernate-table-per-hierarchy-example-using-xml-file)  
[Table Per Hierarchy using Annotation](http://www.javatpoint.com/hibernate-table-per-hierarchy-using-annotation-tutorial-example)

#### Table Per Concrete class

In case of table per concrete class, tables are created as per class. But duplicate column is added in subclass tables.

[Table Per Concrete class using xml file](http://www.javatpoint.com/table-per-concrete-class)  
[Table Per Concrete class using Annotation](http://www.javatpoint.com/hibernate-table-per-concrete-class-using-annotation-tutorial-example)

#### Table Per Subclass

In this strategy, tables are created as per class but related by foreign key. So there are no duplicate columns.

[Table Per Subclass using xml file](http://www.javatpoint.com/table-per-subclass)  
[Table Per Subclass using Annotation](http://www.javatpoint.com/hibernate-table-per-subclass-using-annotation-tutorial-example)

**Sample hibernate configuration file :**

<!DOCTYPE hibernate-configuration PUBLIC

"-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<!--

~ Hibernate, Relational Persistence for Idiomatic Java

~

~ License: GNU Lesser General Public License (LGPL), version 2.1 or later.

~ See the lgpl.txt file in the root directory or <http://www.gnu.org/licenses/lgpl-2.1.html>.

-->

<hibernate-configuration>

<session-factory>

<!-- Database connection settings -->

<property name=*"connection.driver\_class"*>com.mysql.jdbc.Driver</property>

<property name=*"connection.url"*>jdbc:mysql://localhost:3306/hibernatedb</property>

<property name=*"connection.username"*>root</property>

<property name=*"connection.password"*></property>

<!-- JDBC connection pool (use the built-in) -->

<property name=*"connection.pool\_size"*>1</property>

<!-- SQL dialect -->

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

<!-- Enable Hibernate's automatic session context management -->

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

<!-- Disable the second-level cache -->

<property name=*"cache.provider\_class"*>org.hibernate.cache.internal.NoCacheProvider</property>

<!-- Echo all executed SQL to stdout -->

<property name=*"show\_sql"*>true</property>

<!-- Drop and re-create the database schema on startup -->

<property name=*"hbm2ddl.auto"*>create</property>

<mapping class=*"com.sumit.hibernate.model.UserDetails"*/>

<mapping class=*"com.sumit.hibernate.model.Vehicle"*/>

</session-factory>

</hibernate-configuration>

Possible values of hbm2ddl.auto :

* *validate*: validate the schema, makes no changes to the database.
* *update*: update the schema.
* *create*: creates the schema, destroying previous data.
* *create-drop*: drop the schema at the end of the session.

|  |  |
| --- | --- |
| **Annotation** | **Explanation** |
| @Id | annotation marks the identifier for this entity. |
| @GeneratedValue |  |
| @Entity | annotation marks this class as an entity |
| @Table | annotation specifies the table name where data of this entity is to be persisted. If you don't use @Table annotation, hibernate will use the class name as the table name bydefault. |
| @Column | annotation specifies the details of the column for this property or field. If @Column annotation is not specified, property name will be used as the column name bydefault. |
| @Lob | indicates that the property should be persisted in a Blob or a Clob depending on the property type |
| @OnetoOne |  |
| @OneToMany |  |
| @ManyToOne |  |
| @ManyToMany |  |
| @Embeddable |  |
| @Cascade | Cascade is a very convenient feature to manage the state of the other side automatically. However this feature come with a price, if you do not use it wisely (update or delete), it will generate many unnecessary cascade effects (cascade update) to slow down your performance, or delete (cascade delete) some data you didn’t expected. |
| @Inheritance | It defines the inheritance strategy Singletable,joined,table per class |
| @DiscriminatorValue | annotation defines the value used to differentiate a class in the hierarchy. |
| @DiscriminatorColumn |  |
| @Temporal | To save date in different formats |
| @Collection |  |
| @JoinColumn |  |
| @UniqueConstraint | To add unique constraint to the table column. |
| @Embedded | To embed . |
| @AttributeOverrides | To override the name of embedded columns |
| @Basic | annotation allows you to declare the fetching strategy for a property(FetchType.LAZY) |
| @ElementCollection |  |
|  |  |

# Generator classes in Hibernate

# The <generator> subelement of id used to generate the unique identifier for the objects of persistent class. There are many generator classes defined in the Hibernate Framework.

All the generator classes implements the **org.hibernate.id.IdentifierGenerator**[**interface**](http://www.javatpoint.com/interface-in-java). The application programmer may create one's own generator classes by implementing the IdentifierGenerator interface. Hibernate framework provides many built-in generator classes:

1. assigned
2. increment
3. sequence
4. hilo
5. native
6. identity
7. seqhilo
8. uuid
9. guid
10. select
11. foreign
12. sequence-identity

### 1) assigned

It is the default generator strategy if there is no <generator> element . In this case, application assigns the id. For example:

1. ....
2. <hibernate-mapping>
3. <**class** ...>
4. <id ...>
5. <generator **class**="assigned"></generator>
6. </id>
8. .....
10. </**class**>
11. </hibernate-mapping>

### 2) increment

It generates the unique id only if no other process is inserting data into this table. It generates **short**, **int** or **long** type identifier. The first generated identifier is 1 normally and incremented as 1. Syntax:

1. ....
2. <hibernate-mapping>
3. <**class** ...>
4. <id ...>
5. <generator **class**="increment"></generator>
6. </id>
8. .....
10. </**class**>
11. </hibernate-mapping>

### 3) sequence

It uses the sequence of the database. if there is no sequence defined, it creates a sequence automatically e.g. in case of Oracle database, it creates a sequence named HIBERNATE\_SEQUENCE. In case of Oracle, DB2, SAP DB, Postgre SQL or McKoi, it uses sequence but it uses generator in interbase. Syntax:

1. .....
2. <id ...>
3. <generator **class**="sequence"></generator>
4. </id>
5. .....

For defining your own sequence, use the param subelement of generator.

1. .....
2. <id ...>
3. <generator **class**="sequence">
4. <param name="sequence">your\_sequence\_name</param>
5. </generator>
6. </id>
7. .....

### 4) hilo

It uses high and low algorithm to generate the id of type short, int and long. Syntax:

1. .....
2. <id ...>
3. <generator **class**="hilo"></generator>
4. </id>
5. .....

### 5) native

It uses identity, sequence or hilo depending on the database vendor. Syntax:

1. .....
2. <id ...>
3. <generator **class**="native"></generator>
4. </id>
5. .....

### 6) identity

It is used in Sybase, My SQL, MS SQL Server, DB2 and HypersonicSQL to support the id column. The returned id is of type short, int or long.

### 7) seqhilo

It uses high and low algorithm on the specified sequence name. The returned id is of type short, int or long.

### 8) uuid

It uses 128-bit UUID algorithm to generate the id. The returned id is of type String, unique within a network (because IP is used). The UUID is represented in hexadecimal digits, 32 in length.

### 9) guid

|  |
| --- |
| It uses GUID generated by database of type string. It works on MS SQL Server and MySQL. |

### 10) select

|  |
| --- |
| It uses the primary key returned by the database trigger. |

### 11) foreign

|  |
| --- |
| It uses the id of another associated object, mostly used with <one-to-one> association. |

### 12) sequence-identity

|  |
| --- |
| It uses a special sequence generation strategy. It is supported in Oracle 10g drivers only. |

## What are the different levels of ORM quality?

There are four levels defined for ORM quality.

1. Pure relational
2. Light object mapping
3. Medium object mapping
4. Full object mapping

## 5) What is a pure relational ORM?

The entire application, including the user interface, is designed around the relational model and SQL-based relational operations.

## 6) What is a meant by light object mapping?

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

## 7) What is a meant by medium object mapping?

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

## 8)What is meant by full object mapping?

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

# Hibernate SessionFactory openSession vs getCurrentSession vs openStatelessSession

**Hibernate SessionFactory** is the factory class through which we get sessions and perform database operations. Hibernate SessionFactory provides three methods through which we can get Session object –getCurrentSession(), openSession() and openStatelessSession().

### Hibernate 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 like below.

|  |  |
| --- | --- |
| 1 | <property name="hibernate.current\_session\_context\_class">thread</property> |

If its not configured to thread, then we will get below exception.

|  |  |
| --- | --- |
| 1  2  3 | Exception in thread "main" org.hibernate.HibernateException: No CurrentSessionContext configured!      at org.hibernate.internal.SessionFactoryImpl.getCurrentSession(SessionFactoryImpl.java:1012)      at com.journaldev.hibernate.main.HibernateSessionExample.main(HibernateSessionExample.java:16) |

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. Hibernate Session objects are not thread safe, so we should not use it in multi-threaded environment. We can use it in single threaded environment because it’s relatively faster than opening a new session.

### Hibernate openSession

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. For web application frameworks, we can choose to open a new session for each request or for each session based on the requirement.

### Hibernate openStatelessSession

Hibernate SessionFactory openStatelessSession() method returns instance of StatelessSession. There is another overloaded method where we can pass java.sql.Connection object to get a stateless session object from hibernate.

StatelessSession does not implement first-level cache and it doesn’t interact with any second-level cache. Since it’s stateless, it doesn’t implement transactional write-behind or automatic dirty checking or do cascading operations to associated entities.

Collections are also ignored by a stateless session. Operations performed via a stateless session bypass Hibernate’s event model and interceptors. It’s more like a normal JDBC connection and doesn’t provide any benefits that come from using hibernate framework.

However, stateless session can be a good fit in certain situations, for example where we are loading bulk data into database and we don’t want hibernate session to hold huge data in first-level cache memory.

A simple program showing these methods usage is given below.

|  |  |
| --- | --- |
| HibernateSessionExample.java | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37 | package com.journaldev.hibernate.main;    import org.hibernate.Session;  import org.hibernate.SessionFactory;  import org.hibernate.StatelessSession;    import com.journaldev.hibernate.util.HibernateUtil;    public class HibernateSessionExample {        public static void main(String[] args) {            SessionFactory sessionFactory = HibernateUtil.getSessionFactory();            //Current Session - no need to close          Session currentSession = sessionFactory.getCurrentSession();            //open new session          Session newSession = sessionFactory.openSession();          //perform db operations            //close session          newSession.close();            //open stateless session          StatelessSession statelessSession = sessionFactory.openStatelessSession();          //perform stateless db operations            //close session          statelessSession.close();            //close session factory          sessionFactory.close();        }    } |

**Hibernate Merge**

In this section, you will learn about *merge( )* method of Hibernate. Also the Difference between *merge( )* and *update( )* methods.

In Hibernate, a session can not hold two duplicate Persistence Object , means (in a session) two persistence object of same class can not represent the same row. It also assures that two detached object can not represent the same row.

**What is detached object ?**

A new *persistence class* instance, which is not related to a *Session* and has not contained any value representing row of a database table and also have not contained any identifier value, is considered as **transient** by *Hibernate*.

Student student=new Student();

student.setName("Rajesh"); // student is in transient state

While a **persistence instance** , represents a row / rows in the database table and contains an identifier value. Also it is associated with *Session.*

Long id = (Long) session.save(student);

// person is now in a persistent state

You can convert  *transient* instance into *persistence* by associating it with a *Session*.

Now after closing the Hibernate Session, the persistent instance will become a **detached instance or object.**

session.close();

//student is now become detached object or instance

1. **merge( ) method**

As stated above, a session in Hibernate can not hold two duplicate Persistence Object , means means (in a session) two persistence object of same class can not represent the same row. It also assures that two detached object can not represent the same row.

Consider the below code :

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

Student s1 = (Student)session.get(Student .class, 1);

transaction.commit();

session.close();

//s1 is now detached instance.

session = sessionFactory.openSession();

transaction = session.beginTransaction();

Student s2 = (Student )session.get(Student .class, 1);

//As you can see above, s2 represents the same persistent row as s1.

//When we try to reattach s1, an exception is thrown

session.update(s1);

transaction.commit();

session.close();

When you try to execute the above code which attempt to reattach s1 , the following exception occurs :

Exception in thread "main" org.hibernate.NonUniqueObjectException: a

different object with the same identifier value was already associated

with the session: [com.intertech.domain.Student#1]

at

org.hibernate.engine.StatefulPersistenceContext.checkUniqueness

(StatefulPersistenceContext.java:699)

The *merge( )* method is used to deal with situation. After executing below code :

Session session = sessionFactory.openSession();

Transaction transaction = session.beginTransaction();

Student s1 = (Student )session.get(Student .class, 1);

transaction.commit();

session.close();

//s1 is now detached.

session = sessionFactory.openSession();

transaction = session.beginTransaction();

Student s2 = (Student )session.get(Student .class, 1);

Student s3 = (Student )session.merge(s1);

if (s2 == s3) {

System.out.println("s2 and s3 are equal");

}

transaction.commit();

session.close();

you will get the following output :

s2 and s3 are equal

Means if we use *merge( )* method, Hibernate will first check whether a Persistent instance of that type already exists in the persistent  
context. It uses the object identifiers to check on this existence. If another instance exists, it copies the state  
of the Detached object (s1 above) into the existing Persistence object (s2 above). If no other instance exists,  
Hibernate just reattaches the Detached object.

-------------------------------------------------------------------------------------------------

What is difference between session.save() , session.saveOrUpdate() and session.persist()in Hibernate?

session.save() : Save does an insert and will fail if the primary key is already persistent.

session.saveOrUpdate() : saveOrUpdate does a select first to determine if it needs to do an insert or an update. Insert data if primary key not exist otherwise update data.

session.persist() : Does the same like session.save(). But session.save() return Serializable object but session.persist() return void. session.save() returns the generated identifier (Serializable object) and session.persist() doesn't. For Example : if you do :- System.out.println(session.save(question)); This will print the generated primary key. if you do :- System.out.println(session.persist(question)); Compile time error because session.persist() return void.

**Cascade :** In Hibernate the cascade option is used for defining the Owner in the relationship between the associated objects. It is an optional attribute that may be used to specify that which operation should be cascaded. Cascade has the various options "all ", " save | update | delete | merge " that may be used with either of the single option or may be used with the multiple options. Cascade decides the same operation done on the parent object is done on the associated object at the same time. A keyword "cascade" is generally used when there is a mapping does for the collection to handle the state of collection automatically.

**NOTE:** When you will use the cascade with 'all' option it means the owner class has all the manipulation facility that it can apply on the relationship from the parent object to the associated object.

**Example of Cascade:**

A mapping file that contains the mapping of collection mapping of the cascade attribute will be as follows :

<set name="goodsRecords" cascade="all" table="goods\_record">

<key>

<column name="goodsId" not-null="true" />

</key>

<one-to-many class="roseindia.GoodsRecord" />

</set>

**Inverse :** In Hibernate an inverse keyword is used in the hibernate mapping file for maintaining the relationship of the parent and the associated child object. This keyword is responsible for managing the insert / update of the foreign key column. An inverse keyword has the boolean value "true/false". Default value of this keyword is 'false' it means the parent class is responsible for saving/updating the child and it's relationship. And when the keyword inverse is set to the 'true' it means an associated subclass is responsible for saving/updating itself.

NOTE : An inverse keyword is always used with the one-to-many and many-to-many. It doesn't work with many-to-one relationship.

**Example :**

A hibernate mapping file into which the inverse keyword is used as :

<set name="goodsRecords" inverse="true" table="goods\_record">

<key>

<column name="goodsId" not-null="true" />

</key>

<one-to-many class="roseindia.GoodsRecord" />

</set>

The relationship will be converted in owner to owned or vice-versa when you will use the inverse keyword with the following booleanvalue :

if inverse = "true" : In such case according to the above given example goods\_record will be acted as owner and the goods2 will not be updated.

if inverse = "false" : In Hibernate the default value of inverse attribute is set to 'flase' in such case according to the above given example goods2 is the owner and it will update the relationship.

**sorted collection:**

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

**order collection :**

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

**How to call store procedure in Hibernate**

In this tutorial, you will learn how to call a store procedure in Hibernate.

MySQL store procedure

Here’s a MySQL store procedure, which accept a stock code parameter and return the related stock data.

DELIMITER $$

**CREATEPROCEDURE**`GetStocks`(int\_stockcode**VARCHAR**(20))

**BEGIN**

**SELECT**\***FROM** stock **WHERE**stock\_code=int\_stockcode;

**END** $$

DELIMITER ;

In MySQL, you can simple call it with a **call**keyword :

**CALL**GetStocks('7277');

Hibernate call store procedure

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")

**publicclass** 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.

**Hibernate parameter binding examples**

Without parameter binding, you have to concatenate the parameter String like this (bad code) :

Stringhql="from Stock s where s.stockCode = '"+stockCode+"'";

List result =session.createQuery(hql).list();

Pass an unchecked value from user input to the database will raise security concern, because it can easy get hack by SQL injection. You have to avoid the above bad code and using parameter binding instead.

Hibernate parameter binding

There are two ways to parameter binding : named parameters or positional.

1. Named parameters

This is the most common and user friendly way. It use colon followed by a parameter name (:example) to define a named parameter. See examples…

**Example 1 – setParameter**

The **setParameter** is smart enough to discover the parameter data type for you.

Stringhql="from Stock s where s.stockCode= :stockCode";

List result =session.createQuery(hql)

.setParameter("stockCode", "7277")

.list();

**Example 2 – setString**

You can use **setString** to tell Hibernate this parameter date type is String.

Stringhql="from Stock s where s.stockCode= :stockCode";

List result =session.createQuery(hql)

.setString("stockCode", "7277")

.list();

**Example 3 – setProperties**

This feature is great ! You can pass an object into the parameter binding. Hibernate will automatic check the object’s properties and match with the colon parameter.

Stock stock=**new**Stock();

stock.setStockCode("7277");

Stringhql="from Stock s where s.stockCode= :stockCode";

List result =session.createQuery(hql)

.setProperties(stock)

.list();

2. Positional parameters

It’s use question mark (?) to define a named parameter, and you have to set your parameter according to the position sequence. See example…

Stringhql="from Stock s where s.stockCode= ?ands.stockName = ?";

List result =session.createQuery(hql)

.setString(0, "7277")

.setParameter(1, "DIALOG")

.list();

This approach is not support the **setProperties** function. In addition, it’s vulnerable to easy breakage because every change of the position of the bind parameters requires a change to the parameter binding code.

Stringhql="from Stock s where s.stockName= ?ands.stockCode = ?";

List result =session.createQuery(hql)

.setParameter(0, "DIALOG")

.setString(1, "7277")

.list();

Conclusion

In Hibernate parameter binding, i would recommend always go for “**Named parameters**“, as it’s more easy to maintain, and the compiled SQL statement can be reuse (if only bind parameters change) to increase the performance.

**Hibernate native SQL queries examples**

In Hibernate, HQL or criteria queries should be able to let you to execute almost any SQL query you want. However, many developers are complaint about the Hibernate’s generated SQL statement is slow and more prefer to generated their own SQL (native SQL) statement.

Native SQL queries example

Hibernate provide a **createSQLQuery** method to let you call your native SQL statement directly.

1. In this example, you tell Hibernate to return you a Stock.class, all the select data (\*) will match to your Stock.class properties automatically.

Query query=session.createSQLQuery(

"select \* from stock s where s.stock\_code= :stockCode")

.addEntity(Stock.**class**)

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

List result =query.list();

2. In this example, Hibernate will return you an Object array.

Query query=session.createSQLQuery(

"select s.stock\_code from stock s where s.stock\_code= :stockCode")

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

List result =query.list();

Alternative, you also can use the **named query** to call your native SQL statement. See [Hibernate named query examples here](http://www.mkyong.com/hibernate/hibernate-named-query-examples/).

Hibernate’s generated SQL statement is slow !?

I do not agreed on the statement “Hibernate’s generated SQL statement is slow”. Often times, i found out this is because of the developers do not understand the table relationship well, and did some wrong table mappings or misuse the fetch strategies. It will cause Hibernate generated someunnecessary SQL statements or join some unnecessary tables… And developers like to take this excuse and create their own SQL statement to quick fix the bug, and didn’t aware of the core problem will causing more bugs awaiting.

Conclusion

I admit sometime the native SQL statement is really more convenient and easy than HQL, but, do think carefully why you need a native SQL statement? Is this really Hibernate cant do it? If yes, then go ahead ~

*P.S In Oracle database, i more prefer to use native SQL statement in many critical performance queries, because i need to put the “hint” to improve the Oracle query performance.*

**Hibernate named query examples**

By [mkyong](http://www.mkyong.com/author/mkyong/) - February 12, 2010

Often times, developer like to put HQL string literals scatter all over the Java code, this method is hard to maintaine and look ugly. Fortunately, Hibernate come out a technique called “**names queries**” , it lets developer to put all HQL into the XML mapping file or via annotation.

How to declare named query

The named query is supported in both HQL or native SQL. see examples…

1. XML mapping file

HQL in mapping file

*<!-- 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>**

**<query**name="findStockByStockCode"**>**

<![CDATA[from Stock s where s.stockCode = :stockCode]]>

**</query>**

**</hibernate-mapping>**

Native SQL in mapping file

*<!-- 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="findStockByStockCodeNativeSQL"**>**

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

<![CDATA[select \* from stock s where s.stock\_code = :stockCode]]>

**</sql-query>**

**</hibernate-mapping>**

You can place a named query inside ‘**hibernate-mapping**‘ element, but do not put before the ‘**class**‘ element, Hibernate will prompt invalid mapping file, all your named queries have to put after the ‘**class**‘ element.

**Note**  
Regarding the CDATA , it’s always good practice to wrap your query text with CDATA, so that the XML parser will not prompt error for some special XML characters like ‘>’ , <’ and etc.

2. Annotation

HQL in annotation

@NamedQueries({

@NamedQuery(

name="findStockByStockCode",

query="from Stock s where s.stockCode = :stockCode"

)

})

@Entity

@Table(name ="stock", catalog ="mkyong")

**publicclass** Stock **implements**java.io.Serializable{

...

Native SQL in annotation

@NamedNativeQueries({

@NamedNativeQuery(

name="findStockByStockCodeNativeSQL",

query="select \* from stock s where s.stock\_code = :stockCode",

resultClass=Stock.**class**

)

})

@Entity

@Table(name ="stock", catalog ="mkyong")

**publicclass** Stock **implements**java.io.Serializable{

...

In native SQL, you have to declare the ‘**resultClass**‘ to let Hibernate know what is the return type, failed to do it will caused the exception “**org.hibernate.cfg.NotYetImplementedException: Pure native scalar queries are not yet supported**“.

Call a named query

In Hibernate, you can call the named query via **getNamedQuery** method.

Query query=session.getNamedQuery("findStockByStockCode")

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

Query query=session.getNamedQuery("findStockByStockCodeNativeSQL")

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

Conclusion

Named queries are global access, which means the name of a query have to be unique in XML mapping files or annotations. In real environment, it’s always good practice to isolate all the named queries into their own file. In addition, named queries stored in the Hibernate mapping files or annotation are more easier to maintain than queries scattered through the Java code.

What is dynamic-insert

The dynamic-insert attribute tells Hibernate whether to include null properties in the SQL INSERT statement. Let explore some examples to understand more clear about it.

Dynamic-insert example

1. dynamic-insert=false

The default value of dynamic-insert is false, which means **include null properties** in the Hibernate’s SQL INSERT statement.

For example, try set some null values to an object properties and save it.

StockTransactionstockTran=**new**StockTransaction();

*//stockTran.setPriceOpen(new Float("1.2"));*

*//stockTran.setPriceClose(new Float("1.1"));*

*//stockTran.setPriceChange(new Float("10.0"));*

stockTran.setVolume(2000000L);

stockTran.setDate(**new**Date());

stockTran.setStock(stock);

session.save(stockTran);

Turn on the Hibernate “show\_sql” to true, you will see the following insert SQL statement.

Hibernate:

**INSERT**

**INTO**

mkyong.stock\_transaction

(**DATE**, PRICE\_CHANGE, PRICE\_CLOSE, PRICE\_OPEN, STOCK\_ID, VOLUME)

**VALUES**

(?, ?, ?, ?, ?, ?)

**Hibernate will generate the unnecessary columns** (PRICE\_CHANGE, PRICE\_CLOSE, PRICE\_OPEN) for the insertion.

2. dynamic-insert=true

If set the dynamic-insert to true, which means **exclude null property values** in the Hibernate’s SQL INSERT statement.

For example, try set some null values to an object properties and save it again.

StockTransactionstockTran=**new**StockTransaction();

*//stockTran.setPriceOpen(new Float("1.2"));*

*//stockTran.setPriceClose(new Float("1.1"));*

*//stockTran.setPriceChange(new Float("10.0"));*

stockTran.setVolume(2000000L);

stockTran.setDate(**new**Date());

stockTran.setStock(stock);

session.save(stockTran);

Turn on the Hibernate “show\_sql” to true. You will see the different insert SQL statement.

Hibernate:

**INSERT**

**INTO**

mkyong.stock\_transaction

(**DATE**, STOCK\_ID, VOLUME)

**VALUES**

(?, ?, ?)

**Hibernate will generate only the necessary columns** (DATE, STOCK\_ID, VOLUME) for the insertion.

Performance issue

In certain situations, such as a very large table with hundreds of columns (legacy design), or a table contains extremely large data volume, insert something not necessary definitely will drop down your system performance.

How to configure it

You can configure the dynamic-insert properties value through annotation or XML mapping file.

1. Annotation

@Entity

@Table(name ="stock\_transaction", catalog ="mkyong")

@org.hibernate.annotations.Entity(

dynamicInsert=**true**

)

**publicclass**StockTransaction**implements**java.io.Serializable{

2. XML mapping

**<class** ... table="stock\_transaction"catalog="mkyong"dynamic-insert="true"**>**

**<id**name="tranId"type="java.lang.Integer"**>**

**<column**name="TRAN\_ID"**/>**

**<generator**class="identity"**/>**

**</id>**

Conclusion

This little “**dynamic-insert**” tweak may increase your system performance, and highly recommends to do it. However, one question in my mind is why Hibernate set it to false by default?

Dynamic-update example

1. dynamic-update=false

The default value of dynamic-update is false, which means **include unmodified properties** in the Hibernate’s SQL update statement.

For example, get an object and try modify its value and update it.

Query q =session.createQuery("from StockTransaction where tranId = :tranId ");

q.setParameter("tranId", 11);

StockTransactionstockTran=(StockTransaction)q.list().get(0);

stockTran.setVolume(4000000L);

session.update(stockTran);

Hibernate will generate the following update SQL statement.

Hibernate:

**UPDATE**

mkyong.stock\_transaction

**SET**

**DATE**=?,

PRICE\_CHANGE=?,

PRICE\_CLOSE=?,

PRICE\_OPEN=?,

STOCK\_ID=?,

VOLUME=?

**WHERE**

TRAN\_ID=?

Hibernate will update all the unmodified columns.

2. dynamic-update=true

If set the dynamic-insert to true, which means **exclude unmodified properties** in the Hibernate’s SQL update statement.

For example, get an object and try modify its value and update it again.

Query q =session.createQuery("from StockTransaction where tranId = :tranId ");

q.setParameter("tranId", 11);

StockTransactionstockTran=(StockTransaction)q.list().get(0);

stockTran.setVolume(4000000L);

session.update(stockTran);

Hibernate will generate different update SQL statement.

Hibernate:

**UPDATE**

mkyong.stock\_transaction

**SET**

VOLUME=?

**WHERE**

TRAN\_ID=?

Hibernate will update the modified columns only.

**Performance issue**  
In a large table with many columns (legacy design) or contains large data volumes, update some unmodified columns are absolutely unnecessary and great impact on the system performance.

How to configure it

You can configure “dynamic-update” properties via annotation or XML mapping file.

1. Annotation

@Entity

@Table(name ="stock\_transaction", catalog ="mkyong")

@org.hibernate.annotations.Entity(

dynamicUpdate=**true**

)

**publicclass**StockTransaction**implements**java.io.Serializable{

2. XML mapping

**<class** ... table="stock\_transaction"catalog="mkyong"dynamic-update="true"**>**

**<id**name="tranId"type="java.lang.Integer"**>**

**<column**name="TRAN\_ID"**/>**

**<generator**class="identity"**/>**

**</id>**

Conclusion

This little “**dynamic-update**” tweak will definitely increase your system performance, and highly recommended to do it.

**Hibernate – Unable to insert if column named is keyword, such as DESC**

In Hibernate, to insert into “keyword” column name, you should enclose it like this ‘**[column name]**‘.

Hibernate XML mapping file

**<hibernate-mapping>**

**<class**name="com.mkyong.stock.Category"table="category"catalog="mkyongdb"**>**

...

**<property**name="desc"type="string"**>**

**<column**name="[DESC]"not-null="true"**/>**

**</property>**

...

**</class>**

**</hibernate-mapping>**

Or Hibernate annotation

@Column(name ="[DESC]", nullable=**false**)

**public**StringgetDesc(){

**returnthis**.desc;

}

In Hibernate 3.6, “org.hibernate.cfg.AnnotationConfiguration” is deprecated, and all its functionality has been moved to “org.hibernate.cfg.Configuration“.

So , you can safely replace your “**AnnotationConfiguration**” with “**Configuration**” class.

Code snippets …

**import**org.hibernate.cfg.Configuration;

*//...*

**privatestatic**SessionFactorybuildSessionFactory(){

**try**{

**returnnew** Configuration().configure().buildSessionFactory();

}**catch**(Throwable ex){

System.err.println("Initial SessionFactory creation failed."+ex);

**thrownew**ExceptionInInitializerError(ex);

}

}

Create maven project:

mvnarchetype:generate-DgroupId=com.mkyong-DartifactId=HibernateExample

-DarchetypeArtifactId=maven-archetype-quickstart-DinteractiveMode=**false**

<http://www.mkyong.com/hibernate/maven-3-hibernate-3-6-oracle-11g-example-xml-mapping/>

<http://www.mkyong.com/hibernate/how-to-generate-code-with-hibernate-tools/>

**Hibernate mutable example (class and collection)**

In hibernate, ‘**mutable**‘ is default to ‘true’ in class and its related collection, it mean the class or collection are allow to add, update and delete. On the other hand, if the mutable is changed to false, it has different meaning in class and its related collection. Let’s take some examples to understand more about it.

Hibernate one-to-many example

I will take this [one-to-many example](http://www.mkyong.com/hibernate/hibernate-one-to-many-relationship-example/) for the mutable demonstration. In this mapping file, a Stock is belong to many StockDailyRecord.

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

...

**<hibernate-mapping>**

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

**<set**name="stockDailyRecords"mutable="false"cascade="all"

inverse="true"lazy="true"table="stock\_daily\_record"**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="com.mkyong.common.StockDailyRecord"**/>**

**</set>**

**</class>**

...

**</hibernate-mapping>**

How to declare mutable ?

The ‘mutable’ is support both in XML mapping file and annotation.

1. XML mapping file

In mapping file, the ‘**mutable**‘ keyword is use to implement the mutable function.

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

...

**<hibernate-mapping>**

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

**<set**name="stockDailyRecords"mutable="false"cascade="all"

inverse="true"lazy="true"table="stock\_daily\_record"**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="com.mkyong.common.StockDailyRecord"**/>**

**</set>**

**</class>**

...

**</hibernate-mapping>**

2. Annotation

In annotation, the keyword is changed to @Immutable (mutable=’false’).

...

@Entity

@Immutable

@Table(name ="stock", catalog ="mkyong")

**publicclass** Stock **implements**java.io.Serializable{

...

@OneToMany(fetch =FetchType.LAZY, mappedBy="stock")

@Immutable

**public** Set<StockDailyRecord>getStockDailyRecords(){

**returnthis**.stockDailyRecords;

}

...

Mutable in class

If mutable = “false” or @Immutable is declared in class element, it means the **updates to this class will be ignored, but no exception is thrown, only the add and delete operation are allow**.

1. Test insert

Stock stock=**new**Stock();

stock.setStockCode("7277");

stock.setStockName("DIALOG");

session.save(stock);

if mutable = “true” (default) or no @Immutable is declared in class.  
*Output*

Hibernate:

insert into mkyong.stock**(**STOCK\_CODE, STOCK\_NAME**)**

values**(**?, ?**)**

if mutable = “false” or @Immutable is declared in class.  
*Output*

Hibernate:

insert into mkyong.stock**(**STOCK\_CODE, STOCK\_NAME**)**

values**(**?, ?**)**

**Mutable in class has no effect in the ‘insert’ operation.**

2. Test update

Stock stock=(Stock)session.createQuery(

" from Stock where stockCode = '7277'").list().get(0);

stock.setStockName("DIALOG123");

session.saveOrUpdate(stock);

if mutable = “true” or no @Immutable is declared in class.  
*Output*

Hibernate:

**select** ...from mkyong.stock stock0\_

where stock0\_.STOCK\_CODE='7277'

Hibernate:

updatemkyong.stock

**set**STOCK\_CODE=?,STOCK\_NAME=?

whereSTOCK\_ID=?

if mutable = “false” or @Immutable is declared in class.  
*Output*

Hibernate:

**select** ...from mkyong.stock stock0\_

where stock0\_.STOCK\_CODE='7277'

**Mutable in class is not allow application to update it, the ‘update’ operation will be ignore and no exception is thrown**

3. Test delete

Stock stock=(Stock)session.createQuery(

" from Stock where stockCode = '7277'").list().get(0);

session.delete(stock);

if mutable = “true” (default) or no @Immutable is declared in class.  
*Output*

Hibernate:

delete from mkyong.stock

whereSTOCK\_ID=?

if mutable = “false” or @Immutable is declared in class.  
*Output*

Hibernate:

delete from mkyong.stock

whereSTOCK\_ID=?

**Mutable in class has no effect in the ‘delete’ operation.**

Mutable in collection

If mutable = “false” or @Immutable is declared in collection, it means the **add and delete-orphan are not allow in this collection, with exception throw, only update and ‘cascade delete all’ are allow**.

1. Test insert

Assume the cascade insert is enabled.

Stock stock=(Stock)session.createQuery(

" from Stock where stockCode = '7277'").list().get(0);

StockDailyRecordsdr=**new**StockDailyRecord();

sdr.setDate(**new**Date());

sdr.setStock(stock);

stock.getStockDailyRecords().add(sdr);

session.save(stock);

if mutable = “true” (default) or no @Immutable is declared in collection.  
*Output*

Hibernate:

insert into mkyong.stock\_daily\_record

**(**STOCK\_ID, PRICE\_OPEN, PRICE\_CLOSE, PRICE\_CHANGE, VOLUME, DATE**)**

values**(**?, ?, ?, ?, ?, ?**)**

if mutable = “false” or @Immutable is declared in collection.  
*Output*

Exception **in** thread "main"org.hibernate.HibernateException:

changed an immutable collection instance:

**[**com.mkyong.common.Stock.stockDailyRecords*#111]*

**Mutable in collection is not allow the ‘add’ operation, an exception will throw.**

2. Test update

Assume the cascade update is enabled.

Stock stock=(Stock)session.createQuery(

" from Stock where stockCode = '7277'").list().get(0);

StockDailyRecordsdr=stock.getStockDailyRecords().iterator().next();

sdr.setPriceChange(**new**Float(1.30));

session.saveOrUpdate(stock);

if mutable = “true” (default) or no @Immutable is declared in collection.  
*Output*

Hibernate:

updatemkyong.stock\_daily\_record

**set**PRICE\_CHANGE=?, ...

whereDAILY\_RECORD\_ID=?

if mutable = “false” or @Immutable is declared in collection.  
*Output*

Hibernate:

updatemkyong.stock\_daily\_record

**set**PRICE\_CHANGE=?, ...

whereDAILY\_RECORD\_ID=?

**Mutable in collection has no effect in the ‘update’ operation.**

3. Test delete-orphan

Assume the [cascade delete-orphan](http://www.mkyong.com/hibernate/hibernate-cascade-example-save-update-delete-and-delete-orphan/) is enabled.

Stock stock=(Stock)session.createQuery(

" from Stock where stockCode = '7277'").list().get(0);

StockDailyRecordsdr=stock.getStockDailyRecords().iterator().next();

stock.getStockDailyRecords().remove(sdr);

session.saveOrUpdate(stock);

if mutable = “true” (default) or no @Immutable is declared in collection.  
*Output*

Hibernate:

delete from mkyong.stock\_daily\_record

whereDAILY\_RECORD\_ID=?

if mutable = “false” or @Immutable is declared in collection.  
*Output*

Exception **in** thread "main"org.hibernate.HibernateException:

changed an immutable collection instance:

**[**com.mkyong.common.Stock.stockDailyRecords*#111]*

**Mutable in collection is not allow the ‘delete-orphan’ operation, an exception will throw.**

4. Test delete

Assume the cascade delete is enabled.

Stock stock=(Stock)session.createQuery(

" from Stock where stockCode = '7277'").list().get(0);

session.saveOrUpdate(stock);

if mutable = “true” (default) or no @Immutable is declared in collection.  
*Output*

Hibernate:

delete from mkyong.stock\_daily\_record

whereDAILY\_RECORD\_ID=?

Hibernate:

delete from mkyong.stock

whereSTOCK\_ID=?

if mutable = “false” or @Immutable is declared in collection.  
*Output*

Hibernate:

delete from mkyong.stock\_daily\_record

whereDAILY\_RECORD\_ID=?

Hibernate:

delete from mkyong.stock

whereSTOCK\_ID=?

**Mutable in collection has no effect in the ‘delete’ operation, if parent is deleted, all its child will be delete as well, even it’s mutable.**

Why mutable ?

Mutable can avoid many unintentional database operation, like add, update or delete some records which shouldn’t be. In addition, according to Hibernate documentation, the mutable do has some minor performance optimizations, it’s always recommend to analysis your mapping relationship and implement the mutable as needed.

Summary

**1. mutable = “false” or @Immutable is declared in class**

it means the updates to this class will be ignored, but no exception is thrown, only the add and delete operation are allow.

*In Class with mutable=”false” – insert=allow, delete=allow , update=not allow*

**2. mutable = “false” or @Immutable is declared in collection**

it means the add and delete-orphan are not allow in this collection, with exception throw, only update allow. However, if cascade delete is enable, when the parent is deleted, all it’s child will be delete as well, even it is mutable.

*In Collection with mutable=”false” – insert=not allow, delete-orphan=not allow, delete=allow , update=allow*

Completely immutable ?

Can a class completely immutable to any actions? Yes, put a mutable=”false” to all it’s relationship (insert=not allow, delete-orphan=not allow), and a mutable=”false” to the class you want to immutable (update=not allow). Now, you have a completely immutable class, however, if the cascade delete option is enabled, when the parent of your immutable class is deleted, your immutable class will still be deleted as well.

**Hibernate – Cascade example (save, update, delete and delete-orphan)**

Cascade is a convenient feature to save the lines of code needed to manage the state of the other side manually.

The “Cascade” keyword is often appear on the collection mapping to manage the state of the collection automatically. In this tutorials, this [one-to-many example](http://www.mkyong.com/hibernate/hibernate-one-to-many-relationship-example/) will be used to demonstrate the cascade effect.

Cascade save / update example

In this example, if a ‘Stock’ is saved, all its referenced ‘stockDailyRecords’ should be saved into database as well.

1. No save-update cascade

In [previous section](http://www.mkyong.com/hibernate/hibernate-one-to-many-relationship-example/), if you want to save the ‘Stock’ and its referenced ‘StockDailyRecord’ into database, you need to save both individually.

Stock stock=**new**Stock();

StockDailyRecordstockDailyRecords=**new**StockDailyRecord();

*//set the stock and stockDailyRecords data*

stockDailyRecords.setStock(stock);

stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);

session.save(stockDailyRecords);

*Output*

Hibernate:

insert into mkyong.stock**(**STOCK\_CODE, STOCK\_NAME**)**

values**(**?, ?**)**

Hibernate:

insert into mkyong.stock\_daily\_record

**(**STOCK\_ID, PRICE\_OPEN, PRICE\_CLOSE, PRICE\_CHANGE, VOLUME, DATE**)**

values**(**?, ?, ?, ?, ?, ?**)**

2. With save-update cascade

The **cascade=”save-update”** is declared in ‘stockDailyRecords’ to enable the save-update cascade effect.

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

**<set**name="stockDailyRecords"cascade="save-update"table="stock\_daily\_record"...**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="com.mkyong.common.StockDailyRecord"**/>**

**</set>**

Stock stock=**new**Stock();

StockDailyRecordstockDailyRecords=**new**StockDailyRecord();

*//set the stock and stockDailyRecords data*

stockDailyRecords.setStock(stock);

stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);

*Output*

Hibernate:

insert into mkyong.stock**(**STOCK\_CODE, STOCK\_NAME**)**

values**(**?, ?**)**

Hibernate:

insert into mkyong.stock\_daily\_record

**(**STOCK\_ID, PRICE\_OPEN, PRICE\_CLOSE, PRICE\_CHANGE, VOLUME, DATE**)**

values**(**?, ?, ?, ?, ?, ?**)**

The code **session.save(stockDailyRecords);** is no longer required, when you save the ‘Stock’, it will “cascade” the save operation to it’s referenced ‘stockDailyRecords’ and save both into database automatically.

Cascade delete example

In this example, if a ‘Stock’ is deleted, all its referenced ‘stockDailyRecords’ should be deleted from database as well.

1. No delete cascade

You need to loop all the ‘stockDailyRecords’ and delete it one by one.

Query q =session.createQuery("from Stock where stockCode = :stockCode ");

q.setParameter("stockCode", "4715");

Stock stock=(Stock)q.list().get(0);

**for**(StockDailyRecordsdr:stock.getStockDailyRecords()){

session.delete(sdr);

}

session.delete(stock);

*Output*

Hibernate:

delete from mkyong.stock\_daily\_record

whereDAILY\_RECORD\_ID=?

Hibernate:

delete from mkyong.stock

whereSTOCK\_ID=?

2. With delete cascade

The **cascade=”delete”** is declared in ‘stockDailyRecords’ to enable the delete cascade effect. When you delete the ‘Stock’, all its reference ‘stockDailyRecords’ will be deleted automatically.

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

**<set**name="stockDailyRecords"cascade="delete"table="stock\_daily\_record" ...**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="com.mkyong.common.StockDailyRecord"**/>**

**</set>**

Query q =session.createQuery("from Stock where stockCode = :stockCode ");

q.setParameter("stockCode", "4715");

Stock stock=(Stock)q.list().get(0);

session.delete(stock);

*Output*

Hibernate:

delete from mkyong.stock\_daily\_record

whereDAILY\_RECORD\_ID=?

Hibernate:

delete from mkyong.stock

whereSTOCK\_ID=?

Cascade delete-orphan example

In above cascade delete option, if you delete a Stock , all its referenced ‘stockDailyRecords’ will be deleted from database as well. How about if you just want to delete two referenced ‘stockDailyRecords’ records? This is called orphan delete, see example…

1. No delete-orphan cascade

You need to delete the ‘stockDailyRecords’ one by one.

StockDailyRecord sdr1 =(StockDailyRecord)session.get(StockDailyRecord.**class**,

**new**Integer(56));

StockDailyRecord sdr2 =(StockDailyRecord)session.get(StockDailyRecord.**class**,

**new**Integer(57));

session.delete(sdr1);

session.delete(sdr2);

*Output*

Hibernate:

delete from mkyong.stock\_daily\_record

whereDAILY\_RECORD\_ID=?

Hibernate:

delete from mkyong.stock\_daily\_record

whereDAILY\_RECORD\_ID=?

2. With delete-orphan cascade

The **cascade=”delete-orphan”** is declared in ‘stockDailyRecords’ to enable the delete orphan cascade effect. When you save or update the Stock, it will remove those ‘stockDailyRecords’ which already mark as removed.

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

**<set**name="stockDailyRecords"cascade="delete-orphan"table="stock\_daily\_record"**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="com.mkyong.common.StockDailyRecord"**/>**

**</set>**

StockDailyRecord sdr1 =(StockDailyRecord)session.get(StockDailyRecord.**class**,

**new**Integer(56));

StockDailyRecord sdr2 =(StockDailyRecord)session.get(StockDailyRecord.**class**,

**new**Integer(57));

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

stock.getStockDailyRecords().remove(sdr1);

stock.getStockDailyRecords().remove(sdr2);

session.saveOrUpdate(stock);

*Output*

Hibernate:

delete from mkyong.stock\_daily\_record

whereDAILY\_RECORD\_ID=?

Hibernate:

delete from mkyong.stock\_daily\_record

whereDAILY\_RECORD\_ID=?

In short, delete-orphan allow parent table to delete few records (delete orphan) in its child table.

How to enable cascade ?

The cascade is supported in both XML mapping file and annotation.

1. XML mapping file

In XML mapping file, declared the **cascade** keyword in your relationship variable.

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

**<set**name="stockDailyRecords"cascade="save-update, delete"

table="stock\_daily\_record" ...**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="com.mkyong.common.StockDailyRecord"**/>**

**</set>**

2. Annotation

In annotation, declared the **CascadeType.SAVE\_UPDATE** (save, update) and **CascadeType.REMOVE** (delete) in @Cascade annotation.

*//Stock.java*

@OneToMany(mappedBy="stock")

@Cascade({CascadeType.SAVE\_UPDATE, CascadeType.DELETE})

**public** Set<StockDailyRecord>getStockDailyRecords(){

**returnthis**.stockDailyRecords;

}

Further study – [Cascade – JPA & Hibernate annotation common mistake](http://www.mkyong.com/hibernate/cascade-jpa-hibernate-annotation-common-mistake/).

inversevs cascade

Both are totally different notions, see the [differential here](http://www.mkyong.com/hibernate/different-between-cascade-and-inverse/).

Conclusion

Cascade is a very convenient feature to manage the state of the other side automatically. However this feature come with a price, if you do not use it wisely (update or delete), it will generate many unnecessary cascade effects (cascade update) to slow down your performance, or delete (cascade delete) some data you didn’t expected.

**inverse = “true” example and explanation**

**Always put inverse=”true” in your collection variable ?**  
There are many Hibernate articles try to explain the “inverse” with many Hibernate “official” jargon, which is very hard to understand (at least to me). In few articles, they even suggested that just forget about what is “inverse”, and always put **inverse=”true”** in the collection variable.

This statement is always true – “put inverse=true in collection variable”, but do not blindfold on it, try to understand the reason behind is essential to optimal your Hibernate performance.

What is “inverse” ?

This is the most confusing keyword in Hibernate, at least i took quite a long time to understand it. The “**inverse**” keyword is always declare in **one-to-many** and **many-to-many** relationship (many-to-one doesn’t has inverse keyword), it means which side is responsible to take care of the relationship.

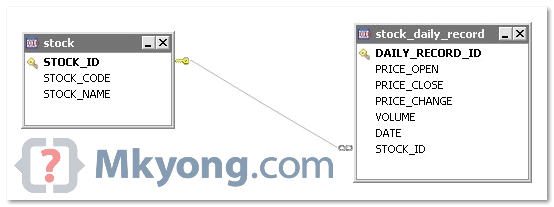
“inverse”, should change to “relationship owner”?

In Hibernate, only the “relationship owner” should maintain the relationship, and the “inverse” keyword is created to defines which side is the owner to maintain the relationship. However the “inverse” keyword itself is not verbose enough, I would suggest change the keyword to “**relationship\_owner**“.

In short, inverse=”true” means this is the relationship owner, and inverse=”false” (default) means it’s not.

1. One to many Relationship

This is a **one-to-many** relationship table design, a STOCK table has many occurrences in STOCK\_DAILY\_RECORD table.



2. Hibernate Implementation

See the Hibernate implementation in XML mapping files.

*File : Stock.java*

**publicclass** Stock **implements**java.io.Serializable{

...

**private** Set<StockDailyRecord>stockDailyRecords=

**new**HashSet<StockDailyRecord>(0);

...

*File : StockDailyRecord.java*

**publicclass**StockDailyRecord**implements**java.io.Serializable{

...

**private** Stock stock;

...

*File : Stock.hbm.xml*

**<hibernate-mapping>**

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

...

**<set**name="stockDailyRecords"table="stock\_daily\_record"fetch="select"**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="com.mkyong.common.StockDailyRecord"**/>**

**</set>**

...

*File : StockDailyRecord.hbm.xml*

**<hibernate-mapping>**

**<class**name="com.mkyong.common.StockDailyRecord"table="stock\_daily\_record" ...**>**

...

**<many-to-one**name="stock"class="com.mkyong.common.Stock"**>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</many-to-one>**

...

3. inverse = true / false

Inverse keyword is applied in one to many relationship. Here’s the question, if save or update operation perform in “Stock” object, should it update the “stockDailyRecords” relationship?

*File : Stock.hbm.xml*

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

...

**<set**name="stockDailyRecords"table="stock\_daily\_record"inverse="{true/false}"fetch="select"**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="com.mkyong.common.StockDailyRecord"**/>**

**</set>**

...

**1. inverse=”true”**

If inverse=”true” in the set variable, it means “stock\_daily\_record” is the relationship owner, so Stock will NOT UPDATE the relationship.

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

...

**<set**name="stockDailyRecords"table="stock\_daily\_record"inverse="true"**>**

**2. inverse=”false”**

If inverse=”false” (default) in the set variable, it means “stock” is the relationship owner, and Stock will UPDATE the relationship.

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

...

**<set**name="stockDailyRecords"table="stock\_daily\_record"inverse="false"**>**

See more examples below :

4. inverse=”false” Example

If keyword “inverse” is not define, the inverse = “false” will be used, which is

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

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

...

**<set**name="stockDailyRecords"table="stock\_daily\_record"inverse="false"**>**

It means “stock” is the relationship owner, and it will maintains the relationship.

**Insert example …**

When a “Stock” object is saved, Hibernate will generated three SQL statements, two inserts and one update.

session.beginTransaction();

Stock stock=**new**Stock();

stock.setStockCode("7052");

stock.setStockName("PADINI");

StockDailyRecordstockDailyRecords=**new**StockDailyRecord();

stockDailyRecords.setPriceOpen(**new**Float("1.2"));

stockDailyRecords.setPriceClose(**new**Float("1.1"));

stockDailyRecords.setPriceChange(**new**Float("10.0"));

stockDailyRecords.setVolume(3000000L);

stockDailyRecords.setDate(**new**Date());

stockDailyRecords.setStock(stock);

stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);

session.save(stockDailyRecords);

session.getTransaction().commit();

*Output…*

Hibernate:

**INSERT**

**INTO**

mkyongdb.stock

(STOCK\_CODE, STOCK\_NAME)

**VALUES**

(?, ?)

Hibernate:

**INSERT**

**INTO**

mkyongdb.stock\_daily\_record

(STOCK\_ID, PRICE\_OPEN, PRICE\_CLOSE, PRICE\_CHANGE, VOLUME,**DATE**)

**VALUES**

(?, ?, ?, ?, ?, ?)

Hibernate:

**UPDATE**

mkyongdb.stock\_daily\_record

**SET**

STOCK\_ID=?

**WHERE**

RECORD\_ID=?

Stock will update the “**stock\_daily\_record.STOCK\_ID**” through Set variable (stockDailyRecords), because Stock is the relationship owner.

**Note**  
The third statement is really NOT necessary.

**Update example …**

When a “Stock” object is updated, Hibernate will generated two SQL statements, one inserts and one update.

session.beginTransaction();

Stock stock=(Stock)session.get(Stock.**class**, 57);

StockDailyRecordstockDailyRecords=**new**StockDailyRecord();

stockDailyRecords.setPriceOpen(**new**Float("1.2"));

stockDailyRecords.setPriceClose(**new**Float("1.1"));

stockDailyRecords.setPriceChange(**new**Float("10.0"));

stockDailyRecords.setVolume(3000000L);

stockDailyRecords.setDate(**new**Date());

stockDailyRecords.setStock(stock);

stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stockDailyRecords);

session.update(stock);

session.getTransaction().commit();

*Output…*

Hibernate:

**INSERT**

**INTO**

mkyongdb.stock\_daily\_record

(STOCK\_ID, PRICE\_OPEN, PRICE\_CLOSE, PRICE\_CHANGE, VOLUME,**DATE**)

**VALUES**

(?, ?, ?, ?, ?, ?)

Hibernate:

**UPDATE**

mkyongdb.stock\_daily\_record

**SET**

STOCK\_ID=?

**WHERE**

RECORD\_ID=?

**Note**  
Again, the third statement is NOT necessary.

5. inverse=”true” Example

If keyword “inverse=true” is defined :

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

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

...

**<set**name="stockDailyRecords"table="stock\_daily\_record"inverse="true"**>**

Now, it means “**stockDailyRecords**” is the relationship owner, and “stock” will not maintains the relationship.

**Insert example …**

When a “Stock” object is saved, Hibernate will generated two SQL insert statements.

session.beginTransaction();

Stock stock=**new**Stock();

stock.setStockCode("7052");

stock.setStockName("PADINI");

StockDailyRecordstockDailyRecords=**new**StockDailyRecord();

stockDailyRecords.setPriceOpen(**new**Float("1.2"));

stockDailyRecords.setPriceClose(**new**Float("1.1"));

stockDailyRecords.setPriceChange(**new**Float("10.0"));

stockDailyRecords.setVolume(3000000L);

stockDailyRecords.setDate(**new**Date());

stockDailyRecords.setStock(stock);

stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stock);

session.save(stockDailyRecords);

session.getTransaction().commit();

*Output …*

Hibernate:

**INSERT**

**INTO**

mkyongdb.stock

(STOCK\_CODE, STOCK\_NAME)

**VALUES**

(?, ?)

Hibernate:

**INSERT**

**INTO**

mkyongdb.stock\_daily\_record

(STOCK\_ID, PRICE\_OPEN, PRICE\_CLOSE, PRICE\_CHANGE, VOLUME,**DATE**)

**VALUES**

(?, ?, ?, ?, ?, ?)

**Update example …**

When a “Stock” object is updated, Hibernate will generated one SQL statement.

session.beginTransaction();

Stock stock=(Stock)session.get(Stock.**class**, 57);

StockDailyRecordstockDailyRecords=**new**StockDailyRecord();

stockDailyRecords.setPriceOpen(**new**Float("1.2"));

stockDailyRecords.setPriceClose(**new**Float("1.1"));

stockDailyRecords.setPriceChange(**new**Float("10.0"));

stockDailyRecords.setVolume(3000000L);

stockDailyRecords.setDate(**new**Date());

stockDailyRecords.setStock(stock);

stock.getStockDailyRecords().add(stockDailyRecords);

session.save(stockDailyRecords);

session.update(stock);

session.getTransaction().commit();

*Output…*

Hibernate:

**INSERT**

**INTO**

mkyongdb.stock\_daily\_record

(STOCK\_ID, PRICE\_OPEN, PRICE\_CLOSE, PRICE\_CHANGE, VOLUME,**DATE**)

**VALUES**

(?, ?, ?, ?, ?, ?)

**inversevs cascade**  
Many people like to compare between inverse and cascade, but both are totally different notions, see the [differential here](http://www.mkyong.com/hibernate/different-between-cascade-and-inverse/).

Conclusion

Understanding the “inverse” is essential to optimize your Hibernate code, it helps to avoid many unnecessary update statements, like “insert and update example for inverse=false” above. At last, try to remember the inverse=”true” mean this is the relationship owner to handle the relationship.

**Different between cascade and inverse**

Many Hibernate developers are confuse about the cascade option and inverse keyword. In some ways..they really look quite similar at the beginning, both are related with relationship.

Cascade vs inverse

However, there is no relationship between cascade and inverse, both are totally different notions.

1. inverse

This is used to decide which side is the relationship owner to manage the relationship (insert or update of the foreign key column).

**Example**

In this example, the relationship owner is belong to stockDailyRecords (inverse=true).

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

**<hibernate-mapping>**

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

...

**<set**name="stockDailyRecords"table="stock\_daily\_record"inverse="true"**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="com.mkyong.common.StockDailyRecord"**/>**

**</set>**

...

When you save or update the stock object

session.save(stock);

session.update(stock);

Hibernate will only insert or update the STOCK table, no update on the foreign key column. [More detail example here…](http://www.mkyong.com/hibernate/inverse-true-example-and-explanation/)

2. cascade

In cascade, after one operation (save, update and delete) is done, it decide whether it need to call other operations (save, update and delete) on another entities which has relationship with each other.

**Example**

In this example, the cascade=”save-update” is declare on stockDailyRecords.

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

**<hibernate-mapping>**

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

...

**<set**name="stockDailyRecords"table="stock\_daily\_record"

cascade="save-update"inverse="true"**>**

**<key>**

**<column**name="STOCK\_ID"not-null="true"**/>**

**</key>**

**<one-to-many**class="com.mkyong.common.StockDailyRecord"**/>**

**</set>**

...

When you save or update the stock object

session.save(stock);

session.update(stock);

It will inserted or updated the record into STOCK table and call another insert or update statement (cascade=”save-update”) on StockDailyRecord. [More detail example here…](http://www.mkyong.com/hibernate/hibernate-cascade-example-save-update-delete-and-delete-orphan/)

Conclusion

In short, the “inverse” is decide which side will update the foreign key, while “cascade” is decide what’s the follow by operation should execute. Both are look quite similar in relationship, but it’s totally two different things. Hibernate developers are worth to spend time to research on it, because misunderstand the concept or misuse it will bring serious performance or data integrity issue in your application.

**What are the most common methods of Hibernate configuration?**

The most common methods of Hibernate configuration are:

\* Programmatic configuration

|  |
| --- |
| Configuration configuration = new Configuration(); |

|  |  |
| --- | --- |
| 15 | configuration.addClass(javabeat.net.hibernate.EmployeeInfo.class); |

|  |  |
| --- | --- |
| 16 | configuration.setProperty("hibernate.dialect", |

|  |  |
| --- | --- |
| 17 | "org.hibernate.dialect.DerbyDialect"); |

|  |  |
| --- | --- |
| 18 | configuration.setProperty("hibernate.connection.url", |

|  |  |
| --- | --- |
| 19 | "jdbc:derby://localhost:1527/SampleDB"); |

|  |  |
| --- | --- |
| 20 | configuration.setProperty("hibernate.connection.username", "root"); |

|  |  |
| --- | --- |
| 21 | configuration.setProperty("hibernate.connection.driver\_class", |

|  |  |
| --- | --- |
| 22 | "org.apache.derby.jdbc.ClientDriver"); |

|  |  |
| --- | --- |
| 23 | configuration.setProperty("hibernate.connection.password", "root"); |

|  |  |
| --- | --- |
| 24 | configuration.setProperty("hibernate.transaction.factory\_class", |

|  |  |
| --- | --- |
| 25 | "org.hibernate.transaction.JDBCTransactionFactory"); |

|  |  |
| --- | --- |
| 26 | configuration.setProperty("hibernate.current\_session\_context\_class", |

|  |  |
| --- | --- |
| 27 | "thread"); |

|  |  |
| --- | --- |
| 28 | configuration.setProperty("hibernate.show\_sql", "true"); |

\* XML configuration (hibernate.cfg.xml)

**What role does the Session interface play in Hibernate?**

The Session interface is the primary interface used by Hibernate applications. It is a single-threaded, short-lived object representing a conversation between the application and the persistent store. It allows you to create query objects to retrieve persistent objects.

Session session = sessionFactory.openSession();

Session interface role:

\* Wraps a JDBC connection

\* Factory for Transaction

\* Holds a mandatory (first-level) cache of persistent objects, used when navigating the object graph or looking up objects by identifier

**What role does the SessionFactory interface play in Hibernate?**

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

SessionFactory sessionFactory = configuration.buildSessionFactory();

**What is the general flow of Hibernate communication with RDBMS?**

The general flow of Hibernate communication with RDBMS is:

\* Load the Hibernate configuration file and create configuration object. It will automatically load all hbm

mapping files

\* Create session factory from configuration object

\* Get one session from this session factory

\* Create HQL Query

\* Execute query to get list containing Java objects

**What is Hibernate Query Language (HQL)?**

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

**How do you map Java Objects with Database tables?**

\* First we need to write Java domain objects (beans with setter and getter). The variables should be same as database columns.

\* Write hbm.xml, where we map java class to table and database columns to Java class variables.

Example:

<hibernate-mapping>

<class name=”com.test.User” table=”user”>

<property column=”USER\_NAME” length=”255″

name=”userName” not-null=”true” type=”java.lang.String”/>

<property column=”USER\_PASSWORD” length=”255″

name=”userPassword” not-null=”true” type=”java.lang.String”/>

</class>

</hibernate-mapping>

**What Does Hibernate Simplify?**

\* Saving and retrieving your domain objects

\* Making database column and table name changes

\* Centralizing pre save and post retrieve logic

\* Complex joins for retrieving related items

\* Schema creation from object model

**How do you define sequence generated primary key in hibernate?**

Using <generator> tag.

Example:-

<id column=”USER\_ID” name=”id” type=”java.lang.Long”>

<generator class=”sequence”>

<param name=”table”>SEQUENCE\_NAME</param>

<generator>

</id>

**What do you mean by Named – SQL query?**

Named SQL queries are defined in the mapping xml document and called wherever required.

Example:

<sql-query name = “empdetails”>

<return alias=”emp” class=”com.test.Employee”/>

SELECT emp.EMP\_ID AS {emp.empid}, emp.EMP\_ADDRESS AS {emp.address},

emp.EMP\_NAME AS {emp.name} FROM Employee EMP WHERE emp.NAME LIKE :name

</sql-query>

Invoke Named Query:

List people = session.getNamedQuery(”empdetails”).setString(”TomBrady”, name)

.setMaxResults(50).list();

**Explain Criteria API?**

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

Example:

List employees = session.createCriteria(Employee.class)

.add(Restrictions.like(”name”, “a%”) )

.add(Restrictions.like(”address”, “Boston”))

.addOrder(Order.asc(”name”) )

.list();

**Define HibernateTemplate?**

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

**What are the benefits does HibernateTemplate provide?**

The benefits of HibernateTemplate are:

\* HibernateTemplate, a Spring Template class simplifies interactions with Hibernate Session.

\* Common functions are simplified to single method calls.

\* Sessions are automatically closed.

\* Exceptions are automatically caught and converted to runtime exceptions.

**How do you switch between relational databases without code changes?**

Using Hibernate SQL Dialects , we can switch databases. Hibernate will generate appropriate hql queries based on the dialect defined.

**If you want to see the Hibernate generated SQL statements on console, what should we do?**

In Hibernate configuration file set as follows:

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

**What are derived properties?**

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

For example:

@Column(name="MONTHLY\_SALARY")

    private float monthlySalary;

@Formula("MONTHLY\_SALARY\*12")

        private float yearlySalary;

**What is component mapping in Hibernate?**

# Component Mapping:

In component mapping, we will map the dependent object as a component. An component is an object that is stored as an value rather than entity reference. This is mainly used if the dependent object doen't have primary key. It is used in case of composition (HAS-A relation), that is why it is termed as component. Let's see the class that have HAS-A relationship.

1. **package** com.javatpoint;
3. **public** **class** Address {
4. **private** String city,country;
5. **private** **int** pincode;
7. //getters and setters
8. }
9. **package** com.javatpoint;
11. **public** **class** Employee {
12. **private** **int** id;
13. **private** String name;
14. **private** Address address;//HAS-A
16. //getters and setters
17. }

Here, address is a dependent object. Hibernate framework provides the facility to map the dependent object as a component. Let's see how can we map this dependent object in mapping file.

1. ...
2. <**class** name="com.javatpoint.Employee" table="emp177">
3. <id name="id">
4. <generator **class**="increment"></generator>
5. </id>
6. <property name="name"></property>
8. <component name="address" **class**="com.javatpoint.Address">
9. <property name="city"></property>
10. <property name="country"></property>
11. <property name="pincode"></property>
12. </component>
14. </**class**>
15. ...

Let's see the data of the emp177 table.



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

**Sorted collection:**

A sorted collection is sorting a collection by utilizing the sorting features provided by the Java collections framework. The sorting occurs in the memory of JVM which running Hibernate, after the data being read from database using java comparator.

If your collection is not large, it will be more efficient way to sort it.

**Order collection:**

Order collection is sorting a collection by specifying the order-by clause for sorting this collection when retrieval.

If your collection is very large, it will be more efficient way to sort it.

**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 Hibernate’s 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.

**How does Hibernate distinguish between transient (i.e. newly instantiated) and detached objects?**

Hibernate uses the version property, if there is one.

If not uses the identifier value. No identifier value means a new object. This does work only for Hibernate managed surrogate keys. Does not work for natural keys and assigned (i.e. not managed by Hibernate) surrogate keys.

Write your own strategy with Interceptor.isUnsaved().

**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?**

1.You must have a default no-argument constructor for your persistent classes and there should be getXXX() (i.e accessor/getter) and setXXX( i.e. mutator/setter) methods for all your persistable instance variables.

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

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

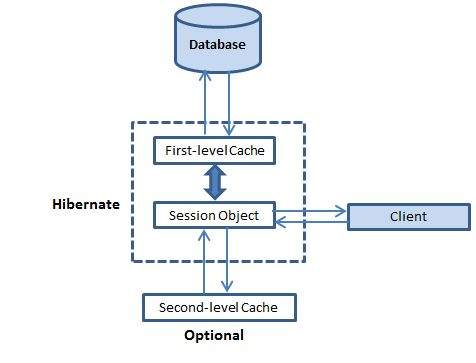
4.The persistent class should not be final because if it is final then lazy loading cannot be used by creating proxy objects.

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

# Hibernate - Caching

Caching is all about application performance optimization and it sits between your application and the database to avoid the number of database hits as many as possible to give a better performance for performance critical applications.

Caching is important to Hibernate as well which utilizes a multilevel caching schemes as explained below:



## First-level cache:

The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database.

If you issue multiple updates to an object, Hibernate tries to delay doing the update as long as possible to reduce the number of update SQL statements issued. If you close the session, all the objects being cached are lost and either persisted or updated in the database.

## Second-level cache:

Second level cache is an optional cache and first-level cache will always be consulted before any attempt is made to locate an object in the second-level cache. The second-level cache can be configured on a per-class and per-collection basis and mainly responsible for caching objects across sessions.

Any third-party cache can be used with Hibernate. An**org.hibernate.cache.CacheProvider** interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation.

## Query-level cache:

Hibernate also implements a cache for query resultsets that integrates closely with the second-level cache.

This is an optional feature and requires two additional physical cache regions that hold the cached query results and the timestamps when a table was last updated. This is only useful for queries that are run frequently with the same parameters.

## The Second Level Cache:

Hibernate uses first-level cache by default and you have nothing to do to use first-level cache. Let's go straight to the optional second-level cache. Not all classes benefit from caching, so it's important to be able to disable the second-level cache

The Hibernate second-level cache is set up in two steps. First, you have to decide which concurrency strategy to use. After that, you configure cache expiration and physical cache attributes using the cache provider.

## 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.

* **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.
* **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.
* **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.
* **Read-only:** A concurrency strategy suitable for data which never changes. Use it for reference data only.

If we are going to use second-level caching for our **Employee** class, let us add the mapping element required to tell Hibernate to cache Employee instances using read-write strategy.

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

<!DOCTYPE hibernate-mapping PUBLIC

"-//Hibernate/Hibernate Mapping DTD//EN"

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

<hibernate-mapping>

<class name="Employee" table="EMPLOYEE">

<meta attribute="class-description">

This class contains the employee detail.

</meta>

<cache usage="read-write"/>

<id name="id" type="int" column="id">

<generator class="native"/>

</id>

<property name="firstName" column="first\_name" type="string"/>

<property name="lastName" column="last\_name" type="string"/>

<property name="salary" column="salary" type="int"/>

</class>

</hibernate-mapping>

The usage="read-write" attribute tells Hibernate to use a read-write concurrency strategy for the defined cache.

## Cache provider:

Your next step after considering the concurrency strategies you will use for your cache candidate classes is to pick a cache provider. Hibernate forces you to choose a single cache provider for the whole application.

|  |  |  |
| --- | --- | --- |
| **S.N.** | **Cache Name** | **Description** |
| 1 | EHCache | It can cache in memory or on disk and clustered caching and it supports the optional Hibernate query result cache. |
| 2 | OSCache | Supports caching to memory and disk in a single JVM, with a rich set of expiration policies and query cache support. |
| 3 | warmCache | A cluster cache based on JGroups. It uses clustered invalidation but doesn't support the Hibernate query cache |
| 4 | JBoss Cache | A fully transactional replicated clustered cache also based on the JGroups multicast library. It supports replication or invalidation, synchronous or asynchronous communication, and optimistic and pessimistic locking. The Hibernate query cache is supported |

Every cache provider is not compatible with every concurrency strategy. The following compatibility matrix will help you choose an appropriate combination.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Strategy/Provider** | **Read-only** | **Nonstrictread-write** | **Read-write** | **Transactional** |
| EHCache | X | X | X |  |
| OSCache | X | X | X |  |
| SwarmCache | X | X |  |  |
| JBoss Cache | X |  |  | X |

You will specify a cache provider in hibernate.cfg.xml configuration file. We choose EHCache as our second-level cache provider:

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

<!DOCTYPE hibernate-configuration SYSTEM

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

<hibernate-configuration>

<session-factory>

<property name="hibernate.dialect">

org.hibernate.dialect.MySQLDialect

</property>

<property name="hibernate.connection.driver\_class">

com.mysql.jdbc.Driver

</property>

<!-- Assume students is the database name -->

<property name="hibernate.connection.url">

jdbc:mysql://localhost/test

</property>

<property name="hibernate.connection.username">

root

</property>

<property name="hibernate.connection.password">

root123

</property>

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

org.hibernate.cache.EhCacheProvider

</property>

<!-- List of XML mapping files -->

<mapping resource="Employee.hbm.xml"/>

</session-factory>

</hibernate-configuration>

Now, you need to specify the properties of the cache regions. EHCache has its own configuration file, **ehcache.xml**, which should be in the CLASSPATH of the application. A cache configuration in ehcache.xml for the Employee class may look like this:

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

<defaultCache

maxElementsInMemory="1000"

eternal="false"

timeToIdleSeconds="120"

timeToLiveSeconds="120"

overflowToDisk="true"

/>

<cache name="Employee"

maxElementsInMemory="500"

eternal="true"

timeToIdleSeconds="0"

timeToLiveSeconds="0"

overflowToDisk="false"

/>

That's it, now we have second-level caching enabled for the Employee class and Hibernate now hits the second-level cache whenever you navigate to a Employee or when you load a Employee by identifier.

You should analyze your all the classes and choose appropriate caching strategy for each of the classes. Sometime, second-level caching may downgrade the performance of the application. So it is recommended to benchmark your application first without enabling caching and later on enable your well suited caching and check the performance. If caching is not improving system performance then there is no point in enabling any type of caching.

## The Query-level Cache:

To use the query cache, you must first activate it using the**hibernate.cache.use\_query\_cache="true"** property in the configuration file. By setting this property to true, you make Hibernate create the necessary caches in memory to hold the query and identifier sets.

Next, to use the query cache, you use the setCacheable(Boolean) method of the Query class. For example:

Session session = SessionFactory.openSession();

Query query = session.createQuery("FROM EMPLOYEE");

query.setCacheable(true);

List users = query.list();

SessionFactory.closeSession();

Hibernate also supports very fine-grained cache support through the concept of a cache region. A cache region is part of the cache that's given a name.

Session session = SessionFactory.openSession();

Query query = session.createQuery("FROM EMPLOYEE");

query.setCacheable(true);

query.setCacheRegion("employee");

List users = query.list();

SessionFactory.closeSession();

This code uses the method to tell Hibernate to store and look for the query in the employee area of the cache.

## Chapter 5. Locking

[5.1. Optimistic](https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html#d0e2225)

[5.1.1. Dedicated version number](https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html#d0e2251)

[5.1.2. Timestamp](https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html#d0e2372)

[5.2. Pessimistic](https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html#d0e2462)

[5.2.1. The LockMode class](https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html#d0e2470)

Locking refers to actions taken to prevent data in a relational database from changing between the time it is read and the time that it is used.

Your locking strategy can be either optimistic or pessimistic.

**Locking strategies**

**Optimistic**

Optimistic locking ssumes that multiple transactions can complete without affecting each other, and that therefore transactions can proceed without locking the data resources that they affect. Before committing, each transaction verifies that no other transaction has modified its data. If the check reveals conflicting modifications, the committing transaction rolls back[[1](https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/ch05.html#ftn.d0e2213)].

**Pessimistic**

Pessimistic locking assumes that concurrent transactions will conflict with each other, and requires resources to be locked after they are read and only unlocked after the application has finished using the data.

Hibernate provides mechanisms for implementing both types of locking in your applications.

## 5.1. Optimistic

When your application uses long transactions or conversations that span several database transactions, you can store versioning data, so that if the same entity is updated by two conversations, the last to commit changes is informed of the conflict, and does not override the other conversation's work. This approach guarantees some isolation, but scales well and works particularly well in Read-Often Write-Sometimessituations.

Hibernate provides two different mechanisms for storing versioning information, a dedicated version number or a timestamp.

**Version number**

**Timestamp**

## Note

A version or timestamp property can never be null for a detached instance. Hibernate detects any instance with a null version or timestamp as transient, regardless of other unsaved-value strategies that you specify. Declaring a nullable version or timestamp property is an easy way to avoid problems with transitive reattachment in Hibernate, especially useful if you use assigned identifiers or composite keys.

### 5.1.1. Dedicated version number

The version number mechanism for optimistic locking is provided through a @Version annotation.

**Example 5.1. The @Version annotation**

@Entity

**public** **class** Flight **implements** Serializable {

...

    @Version

    @Column(name="OPTLOCK")

**public** Integer getVersion() { ... }

}

Here, the version property is mapped to the OPTLOCK column, and the entity manager uses it to detect conflicting updates, and prevent the loss of updates that would be overwritten by a last-commit-winsstrategy.

The version column can be any kind of type, as long as you define and implement the appropriateUserVersionType.

Your application is forbidden from altering the version number set by Hibernate. To artificially increase the version number, see the documentation for properties LockModeType.OPTIMISTIC\_FORCE\_INCREMENT orLockModeType.PESSIMISTIC\_FORCE\_INCREMENTcheck in the Hibernate Entity Manager reference documentation.

## Database-generated version numbers

If the version number is generated by the database, such as a trigger, use the annotation@org.hibernate.annotations.Generated(GenerationTime.ALWAYS).

**Example 5.2. Declaring a version property in hbm.xml**

<version

**column**="version\_column"

**name**="propertyName"

**type**="typename"

**access**="field|property|ClassName"

**unsaved-value**="null|negative|undefined"

**generated**="never|always"

**insert**="true|false"

**node**="element-name|@attribute-name|element/@attribute|."

/>

|  |  |
| --- | --- |
| column | The name of the column holding the version number. Optional, defaults to the property name. |
| name | The name of a property of the persistent class. |
| type | The type of the version number. Optional, defaults to integer. |
| access | Hibernate's strategy for accessing the property value. Optional, defaults to property. |
| unsaved-value | Indicates that an instance is newly instantiated and thus unsaved. This distinguishes it from detached instances that were saved or loaded in a previous session. The default value,undefined, indicates that the identifier property value should be used. Optional. |
| generated | Indicates that the version property value is generated by the database. Optional, defaults tonever. |
| insert | Whether or not to include the version column in SQL insert statements. Defaults to true, but you can set it to false if the database column is defined with a default value of 0. |

### 5.1.2. Timestamp

Timestamps are a less reliable way of optimistic locking than version numbers, but can be used by applications for other purposes as well. Timestamping is automatically used if you the @Version annotation on a Date orCalendar.

**Example 5.3. Using timestamps for optimistic locking**

@Entity

**public** **class** Flight **implements** Serializable {

...

    @Version

**public** Date getLastUpdate() { ... }

}

Hibernate can retrieve the timestamp value from the database or the JVM, by reading the value you specify for the @org.hibernate.annotations.Source annotation. The value can be eitherorg.hibernate.annotations.SourceType.DB or org.hibernate.annotations.SourceType.VM. The default behavior is to use the database, and is also used if you don't specify the annotation at all.

The timestamp can also be generated by the database instead of Hibernate, if you use the@org.hibernate.annotations.Generated(GenerationTime.ALWAYS) annotation.

**Example 5.4. The timestamp element in hbm.xml**

<timestamp

**column**="timestamp\_column"

**name**="propertyName"

**access**="field|property|ClassName"

**unsaved-value**="null|undefined"

**source**="vm|db"

**generated**="never|always"

**node**="element-name|@attribute-name|element/@attribute|."

/>

|  |  |
| --- | --- |
| column | The name of the column which holds the timestamp. Optional, defaults to the property namel |
| name | The name of a JavaBeans style property of Java type Date or Timestamp of the persistent class. |
| access | The strategy Hibernate uses to access the property value. Optional, defaults to property. |
| unsaved-value | A version property which indicates than instance is newly instantiated, and unsaved. This distinguishes it from detached instances that were saved or loaded in a previous session. The default value of undefined indicates that Hibernate uses the identifier property value. |
| source | Whether Hibernate retrieves the timestamp from the database or the current JVM. Database-based timestamps incur an overhead because Hibernate needs to query the database each time to determine the incremental next value. However, database-derived timestamps are safer to use in a clustered environment. Not all database dialects are known to support the retrieval of the database's current timestamp. Others may also be unsafe for locking, because of lack of precision. |
| generated | Whether the timestamp property value is generated by the database. Optional, defaults tonever. |

## 5.2. Pessimistic

Typically, you only need to specify an isolation level for the JDBC connections and let the database handle locking issues. If you do need to obtain exclusive pessimistic locks or re-obtain locks at the start of a new transaction, Hibernate gives you the tools you need.

## Note

Hibernate always uses the locking mechanism of the database, and never lock objects in memory.

### 5.2.1. The LockMode class

The LockMode class defines the different lock levels that Hibernate can acquire.

|  |  |
| --- | --- |
| LockMode.WRITE | acquired automatically when Hibernate updates or inserts a row. |
| LockMode.UPGRADE | acquired upon explicit user request using SELECT ... FOR UPDATE on databases which support that syntax. |
| LockMode.UPGRADE\_NOWAIT | acquired upon explicit user request using a SELECT ... FOR UPDATE NOWAITin Oracle. |
| LockMode.READ | acquired automatically when Hibernate reads data under Repeatable Read orSerializable isolation level. It can be re-acquired by explicit user request. |
| LockMode.NONE | The absence of a lock. All objects switch to this lock mode at the end of a Transaction. Objects associated with the session via a call to update() orsaveOrUpdate() also start out in this lock mode. |

The explicit user request mentioned above occurs as a consequence of any of the following actions:

* A call to Session.load(), specifying a LockMode.
* A call to Session.lock().
* A call to Query.setLockMode().

If you call Session.load() with option **UPGRADE** or **UPGRADE\_NOWAIT**, and the requested object is not already loaded by the session, the object is loaded using SELECT ... FOR UPDATE. If you call load() for an object that is already loaded with a less restrictive lock than the one you request, Hibernate calls lock() for that object.

Session.lock() performs a version number check if the specified lock mode is READ, UPGRADE, orUPGRADE\_NOWAIT. In the case of UPGRADE or UPGRADE\_NOWAIT, SELECT ... FOR UPDATE syntax is used.

If the requested lock mode is not supported by the database, Hibernate uses an appropriate alternate mode instead of throwing an exception. This ensures that applications are portable.