1. Difference between load() and get() methods

Why we have two method to do the same job. Actually this is frequently asked interview question as well.

The difference between get and load methods lies in return value when the identifier does not exist in database.

- In case of **get()** method, we will get return value as **NULL** if identifier is absent.
- But in case of **load()** method, we will get a runtime exception.

The exception in case of load method will look like this:

Exception in thread "main" org.hibernate.ObjectNotFoundException: No row with the given identifier exists:
[com.howtodoinjava.demo.entity.EmployeeEntity#23]
at org.hibernate.internal.SessionFactoryImpl$1$1.handleEntityNotFound(SessionFactoryImpl.java:253)
at org.hibernate.event.internal.DefaultLoadEventListener.load(DefaultLoadEventListener.java:219)
at org.hibernate.event.internal.DefaultLoadEventListener.proxyOrLoad(DefaultLoadEventListener.java:275)
at org.hibernate.event.internal.DefaultLoadEventListener.onLoad(DefaultLoadEventListener.java:151)
at org.hibernate.internal.SessionImpl.fireLoad(SessionImpl.java:1070)
at org.hibernate.internal.SessionImpl.load(SessionImpl.java:940)

2. Hibernate Merging and Refreshing Entities

Refreshing Hibernate Entities Using refresh() Method

Sometimes we face situation where we application database is modified with some external application/agent and thus corresponding hibernate entity in your application actually becomes out of sync with it's database representation i.e. having old data. In this case, you can use **session.refresh()** method to re-populate the entity with latest data available in database.

You can use one of the refresh() methods on the Session interface to refresh an instance of a persistent object, as follows:

public void refresh(Object object) throws HibernateException
public void refresh(Object object, LockMode lockMode) throws HibernateException

These methods will reload the properties of the object from the database, overwriting them. In real life applications, you do not have to use the refresh() method very often apart from above stated scenario.

Let's look at an example of refresh() method.

```
public class RefreshEntityExample {
  public static void main(String[] args) {
    Session sessionOne = HibernateUtil.getSessionFactory().openSession();
    sessionOne.beginTransaction();

    //Create new Employee object
    EmployeeEntity emp = new EmployeeEntity();
    emp.setEmployeeId(1);
```

```
        emp.setFirstName("Lokesh");
        emp.setLastName("Gupta");

        //Save employee
        sessionOne.save(emp);
        sessionOne.getTransaction().commit();
        sessionOne.close();

        //Verify employee's firstname
        System.out.println(verifyEmployeeFirstName(1, "Lokesh"));

        Session sessionTwo = HibernateUtil.getSessionFactory().openSession();
        sessionTwo.beginTransaction();

        //This
        emp.setFirstName("Vikas");
        sessionTwo.refresh(emp);

        sessionTwo.getTransaction().commit();
        sessionTwo.close();

        System.out.println(emp.getFirstName().equals("Lokesh"));

        HibernateUtil.shutdown();
    }

    private static boolean verifyEmployeeFirstName(Integer employeeId, String firstName){
        Session session = HibernateUtil.getSessionFactory().openSession();
        EmployeeEntity employee = (EmployeeEntity) session.load(EmployeeEntity.class, employeeId);
        //Verify first name
        boolean result = firstName.equals(employee.getFirstName());
        session.close();
        //Return verification result
        return result;
    }
}
```

Output:
true
true

Look above the highlighted lines.

- Line 15 save the employee with first name “Lokesh”
- Line 26 set the first name “Vikas”. As entity is detached, DB will not be updated.
- Line 27 refresh the entity with database using refresh() method.
- Line 32 verify that firstname set in entity has been updated with “Lokesh” as it is what database have this moment.

This was all about refresh method. Let’s look an another similar method merge().

Merging Hibernate Entities Using merge() Method
Method merge() does exactly opposite to what refresh() does i.e. It updates the database with values from a detached entity. Refresh method was updating the entity with latest database information. So basically, both are exactly opposite.

Merging is performed when you desire to have a detached entity changed to persistent state again, with the detached entity's changes migrated to (or overriding) the database. The method signatures for the merge operations are:

Object merge(Object object)
Object merge(String entityName, Object object)

Hibernate official documentation give a very good explanation of merge() method:
Copy the state of the given object onto the persistent object with the same identifier. If there is no persistent instance currently associated with the session, it will be loaded. Return the persistent instance. If the given instance is unsaved, save a copy of and return it as a newly persistent instance. The given instance does not become associated with the session. This operation cascades to associated instances if the association is mapped with cascade="merge".

So if I take below code for example then below listed points should be clear to you.

EmployeeEntity mergedEmpEntity = session.merge(empEntity);

- 'empEntity' is detached entity when it is passed to merge() method.
- merge() method will search for an already loaded EmployeeEntity instance with identifier information taken from empEntity. If such persistent entity is found then it will be used for updates. Otherwise a new EmployeeEntity is loaded into session using same identifier information as present in 'empEntity'.
- Data is copied from 'empEntity' to new found/loaded entity.
- Because new/found entity is persistent, all data copied to it from 'empEntity' is automatically saved into database.
- Reference of that new entity is returned from merge() method and is assigned to 'mergedEmpEntity' variable.
- 'empEntity' is still detached entity.

```
public class MergeEntityExample {
  public static void main(String[] args) {
    Session sessionOne = HibernateUtil.getSessionFactory().openSession();
    sessionOne.beginTransaction();

    //Create new Employee object
    EmployeeEntity emp = new EmployeeEntity();
    emp.setEmployeeId(1);
    emp.setFirstName("Lokesh");
    emp.setLastName("Gupta");

    //Save employee
    sessionOne.save(emp);
    sessionOne.getTransaction().commit();
    sessionOne.close();
```

```
    //Verify employee's firstname
    System.out.println(verifyEmployeeFirstName(1, "Lokesh"));

    Session sessionTwo = HibernateUtil.getSessionFactory().openSession();
    sessionTwo.beginTransaction();

    //Set new first name
    emp.setFirstName("Vikas");

    //Merge the emp object using merge() method
    EmployeeEntity mergedPersistentEmpEntity = (EmployeeEntity) sessionTwo.merge(emp);

    sessionTwo.getTransaction().commit();
    sessionTwo.close();

    //Verify employee's firstname again in database
    System.out.println(verifyEmployeeFirstName(1, "Vikas"));

    HibernateUtil.shutdown();
  }

  private static boolean verifyEmployeeFirstName(Integer employeeId, String firstName){
    Session session = HibernateUtil.getSessionFactory().openSession();
    EmployeeEntity employee = (EmployeeEntity) session.load(EmployeeEntity.class, employeeId);
    //Verify first name
    boolean result = firstName.equals(employee.getFirstName());
    session.close();
    //Return verification result
    return result;
  }
}
```

Output:
true
true

In above example, ‘mergedPersistentEmpEntity’ is new entity which is persistent. So if you want to any more change, then make in in ‘mergedPersistentEmpEntity’ instance.

Hibernate 4 – Get Entity Reference for Lazy Loading

By Wikipedia definition, Lazy loading is a design pattern commonly used in computer programming to defer initialization of an object until the point at which it is needed. We know that in hibernate lazy loading can be done by specifying "**fetch= FetchType.LAZY**" in hibernate mapping annotations. e.g.

```
@ManyToOne ( fetch = FetchType.LAZY )
@JoinColumns( {
                @JoinColumn(name="fname", referencedColumnName = "firstname"),
                @JoinColumn(name="lname", referencedColumnName = "lastname")
                } )
public EmployeeEntity getEmployee() {
        return employee;
}
```

The point is that it is applied only when you are defining mapping between two entities. If above entity has been defined in DepartmentEntity then if you fetch DepartmentEntity then EmployeeEntity will be lazy loaded.

But, what if you want to lazy load DepartmentEntity itself i.e. master entity itself should be lazy loaded.

This problem can be solved by using **getReference()** method inside IdentifierLoadAccess class.

Let's understand the usage by this example.

For reference, latest hibernate maven dependency is as follows:

```
<dependency>
        <groupId>org.hibernate</groupId>
        <artifactId>hibernate-core</artifactId>
        <version>4.3.0.Beta3</version>
</dependency>
```

Now, I have an master entity class EmployeeEntity which can have multiple attributes and mapping with other entities.

EmployeeEntity.java

```
@Entity
@Table(name = "Employee", uniqueConstraints = {
                @UniqueConstraint(columnNames = "ID"),
                @UniqueConstraint(columnNames = "EMAIL") })
public class EmployeeEntity implements Serializable {

        private static final long serialVersionUID = -1798070786993154676L;

        @Id
        @GeneratedValue(strategy = GenerationType.IDENTITY)
        @Column(name = "ID", unique = true, nullable = false)
        private Integer employeeId;
```

```
	//Use the natural id annotation here
	@NaturalId (mutable = false)
	@Column(name = "EMAIL", unique = true, nullable = false, length = 100)
	private String email;

	@Column(name = "FIRST_NAME", unique = false, nullable = false, length = 100)
	private String firstName;

	@Column(name = "LAST_NAME", unique = false, nullable = false, length = 100)
	private String lastName;

	//Setters and Getters
}
```

I want to lazy load above master entity lazy loaded in my code i.e. I can get reference of entity in one place but might be actually needing it another place. Only when I need it, I want to initialize or load its data. Till the time, I want only the reference.

Let's do this in code example:

TestHibernate.java

```
public class TestHibernate
{
	public static void main(String[] args)
	{
		Session session = HibernateUtil.getSessionFactory().openSession();
		session.beginTransaction();

		//Add new Employee object
		EmployeeEntity emp = new EmployeeEntity();
		emp.setEmail("demo-user@mail.com");
		emp.setFirstName("demo");
		emp.setLastName("user");
		//Save entity
		session.save(emp);

		session.getTransaction().commit();
		session.close();

		session = HibernateUtil.getSessionFactory().openSession();
		session.beginTransaction();

		//Get only the reference of EmployeeEntity for now
		EmployeeEntity empGet = (EmployeeEntity) session.byId( EmployeeEntity.class
).getReference( 1 );

		System.out.println("No data initialized till now; Lets fetch some data..");
```

```
            //Now EmployeeEntity will be loaded from database when we need it
            System.out.println(empGet.getFirstName());
            System.out.println(empGet.getLastName());

            session.getTransaction().commit();
            HibernateUtil.shutdown();
        }
}
```

Output in console:

Hibernate: insert into Employee (EMAIL, FIRST_NAME, LAST_NAME) values (?, ?, ?)

No data initialized till now; Lets fetch some data..

Hibernate: select employeeen0_.ID as ID1_0_0_, employeeen0_.EMAIL as EMAIL2_0_0_, employeeen0_.FIRST_NAME as FIRST3_0_0_,
employeeen0_.LAST_NAME as LAST4_0_0_ from Employee employeeen0_ where employeeen0_.ID=?

demo
user

Hibernate save() and saveOrUpdate() methods

Please remember that if you have used saveOrUpdate() method in place of save() method above, then also result would have been same. saveOrUpdate() can be used with persistent as well as non-persistent entities both. Persistent entities will get updated, and transient entities will be inserted into database

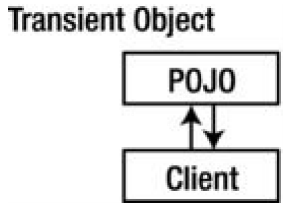
- **Save()** method stores an object into the database. It will Persist the given transient instance, first assigning a generated identifier. It returns the id of the entity created.
- **SaveOrUpdate()** calls either **save()** or **update()** on the basis of identifier exists or not. e.g if identifier does not exist, **save()** will be called or else **update()** will be called.
- Probably you will get very few chances to actually call **save()** or **saveOrUpdate()** methods, as hibernate manages all changes done in persistent objects.

Hibernate Entity / Persistence LifeCycle States

Given an instance of an object that is mapped to Hibernate, it can be in any one of four different states: **transient**, **persistent**, **detached**, or **removed**. We are going to learn about them today in this tutorial.

Transient Object

Transient objects exist in heap memory. Hibernate does not manage transient objects or persist changes to transient objects.

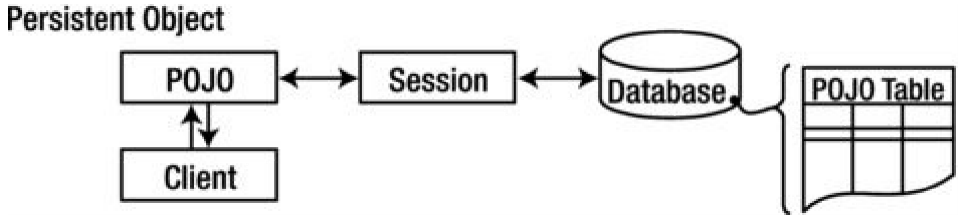


Transient objects are independent of Hibernate

To persist the changes to a transient object, you would have to ask the session to save the transient object to the database, at which point Hibernate assigns the object an identifier and marks the object as being in persistent state.

Persistent Object

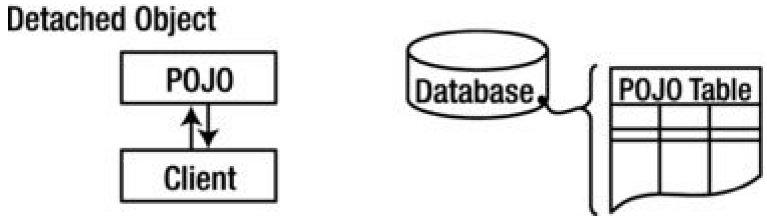
Persistent objects exist in the database, and Hibernate manages the persistence for persistent objects.



If fields or properties change on a persistent object, Hibernate will keep the database representation up to date when the application marks the changes as to be committed.

Detached Object

Detached objects have a representation in the database, but changes to the object will not be reflected in the database, and vice-versa. This temporary separation of the object and the database is shown in image below.



Detached objects exist in the database but are not maintained by Hibernate

A detached object can be created by closing the session that it was associated with, or by evicting it from the session with a call to the session's evict() method.

In order to persist changes made to a detached object, the application must reattach it to a valid Hibernate session. A detached instance can be associated with a new Hibernate session when your application calls one of the load, refresh, merge, update(), or save() methods on the new session with a reference to the detached object. After the call, the detached object would be a persistent object managed by the new Hibernate session.

Removed Object

Removed objects are objects that are being managed by Hibernate (persistent objects, in other words) that have been passed to the session's remove() method. When the application marks the changes held in the session as to be committed, the entries in the database that correspond to removed objects are deleted.

Now let's not note down the take-aways from this tutorial.
Bullet Points

- Newly created POJO object will be in the **transient state**. Transient object doesn't represent any row of the database i.e. not associated with any session object. It's plain simple java object.
- **Persistent object** represent one row of the database and always associated with some unique hibernate session. Changes to persistent objects are tracked by hibernate and are saved into database when commit call happen.
- **Detached objects** are those who were once persistent in past, and now they are no longer persistent. To persist changes done in detached objects, you must reattach them to hibernate session.
- **Removed objects** are persistent objects that have been passed to the session's remove() method and soon will be deleted as soon as changes held in the session will be committed to database.

Hibernate Entities Equality and Identity

Bullet Points

- Requesting a persistent object again from the same Hibernate session returns the "same java instance" of a class.
- Requesting a persistent object from the different Hibernate session returns "different java instance" of a class.
- As a best practice, always implement equals() and hashCode() methods in your hibernate entities; and always compare them using equals() method only.

Hibernate JPA Cascade Types

We learned about mapping associated entities in hibernate already in previous tutorials such as one-to-one mapping and one-to-many mappings. There we wanted to save the mapped entity whenever relationship owner entity got saved. To enable this we had use "CascadeType" attribute. In this JPA Cascade Types tutorial, we will learn about various type of available options for cascading via CascadeType.

JPA Cascade Types

The cascade types supported by the Java Persistence Architecture are as below:

- **CascadeType.PERSIST** : cascade type presist means that save() or persist() operations cascade to related entities.
- **CascadeType.MERGE :** cascade type merge means that related entities are merged when the owning entity is merged.
- **CascadeType.REFRESH :** cascade type refresh does the same thing for the refresh() operation.
- **CascadeType.REMOVE :** cascade type remove removes all related entities association with this setting when the owning entity is deleted.
- **CascadeType.DETACH :** cascade type detach detaches all related entities if a "manual detach" occurs.
- **CascadeType.ALL** : cascade type all is shorthand for all of the above cascade operations.

There is no default cascade type in JPA. By default no operations are cascaded.

The cascade configuration option accepts an array of CascadeTypes; thus, to include only refreshes and merges in the cascade operation for a One-to-Many relationship as in our example, you might see the following:

@OneToMany(**cascade={CascadeType.REFRESH, CascadeType.MERGE},** fetch = FetchType.LAZY)
@JoinColumn(name="EMPLOYEE_ID")
private Set<AccountEntity> accounts;

Above cascading will cause accounts collection to be only merged and refreshed.

Hibernate Cascade Types

Now lets understand what is cascade in hibernate in which scenario we use it.

Apart from JPA provided cascade types, there is one more cascading operation in hibernate which is not part of the normal set above discussed, called **"orphan removal".** This removes an owned object from the database when it's removed from its owning relationship.

Let's understand with an example. In our Employee and Account entity example, I have updated them as below and have mentioned **"orphanRemoval = true"** on accounts. It essentially means that whenever I will remove an 'account from accounts set' (which means I am removing the relationship between that account and Employee); the account entity which is not associated with any other Employee on database (i.e. orphan) should also be deleted.

EmployeeEntity.java

```
@Entity
@Table(name = "Employee")
public class EmployeeEntity implements Serializable
{
	private static final long serialVersionUID = -1798070786993154676L;
	@Id
	@Column(name = "ID", unique = true, nullable = false)
	private Integer		employeeId;
	@Column(name = "FIRST_NAME", unique = false, nullable = false, length = 100)
	private String		firstName;
	@Column(name = "LAST_NAME", unique = false, nullable = false, length = 100)
```

```
    private String          lastName;

    @OneToMany(orphanRemoval = true, mappedBy = "employee")
    private Set<AccountEntity> accounts;

}
```

Hibernate Lazy Loading Tutorial

In any application, hibernate fetches data from databse either in eager or lazy mode. Hibernate lazy loading refer to strategy when data is loaded lazily, on demand.

1. Hibernate lazy loading – why we need it?

Consider one of common Internet web application: the online store. The store maintains a catalog of products. At the crudest level, this can be modeled as a catalog entity managing a series of product entities. In a large store, there may be tens of thousands of products grouped into various overlapping categories.

When a customer visits the store, the catalog must be loaded from the database. We probably don't want the implementation to load every single one of the entities representing the tens of thousands of products to be loaded into memory. For a sufficiently large retailer, this might not even be possible, given the amount of physical memory available on the machine.

Even if this was possible, it would probably cripple the performance of the site. Instead, we want only the catalog to load, possibly with the categories as well. Only when the user drills down into the categories should a subset of the products in that category be loaded from the database.

To manage this problem, Hibernate provides a facility called lazy loading. When enabled, an entity's associated entities will be loaded only when they are directly requested.

2. How lazy loading solve above problem

Now when we have understood the problem, let's understand how lazy loading actually helps in real life. If we consider to solve the problem discussed above then we would be accessing a category (or catalog) in below manner:

```
//Following code loads only a single category from the database:
Category category = (Category)session.get(Category.class,new Integer(42));
```

However, if all products of this category are accessed, and lazy loading is in effect, the products are pulled from the database as needed. For instance, in the following snippet, the associated product objects will be loaded since it is explicitly referenced in second line.

```
//Following code loads only a single category from the database
Category category = (Category)session.get(Category.class,new Integer(42));

//This code will fetch all products for category 42 from database - "NOW"
Set<Product> products = category.getProducts();
```

This solve our problem of loading the products only when they are needed.

3. How to enable lazy loading in hibernate

Before moving further, it is important to recap the default behavior of lazy loading in case of using hibernate mappings vs annotations.

The default behavior is to load 'property values eagerly' and to load 'collections lazily'. Contrary to what you might remember if you have used plain Hibernate 2 (mapping files) before, where all references (including collections) are loaded eagerly by default.

Also note that @OneToMany and @ManyToMany associations are defaulted to LAZY loading; and @OneToOne and @ManyToOne are defaulted to EAGER loading. This is important to remember to avoid any pitfall in future.

To enable lazy loading explicitly you must use "fetch = FetchType.LAZY" on a association which you want to lazy load when you are using hibernate annotations.

A hibernare lazy load example will look like this:

```
@OneToMany( mappedBy = "category", fetch = FetchType.LAZY )
private Set<ProductEntity> products;
```

Another attribute parallel to "FetchType.LAZY" is "FetchType.EAGER" which is just opposite to LAZY i.e. it will load association entity as well when owner entity is fetched first time.

4. How lazy loading works in hibernate

The simplest way that Hibernate can apply lazy load behavior upon the entities and associations is by providing a proxy implementation of them. Hibernate intercepts calls to the entity by substituting a proxy for it derived from the entity's class. Where the requested information is missing, it will be loaded from the database before control is ceded to the parent entity's implementation.

Please note that when the association is represented as a collection class, then a wrapper (essentially a proxy for the collection, rather than for the entities that it contains) is created and substituted for the original collection. When you access this collection proxy then what you get inside returned proxy collection are not proxy entities; rather they are actual entities. You need not to put much pressure on understanding this concept because on runtime it hardly matters.

5. Effect of lazy loading on detached entities

As we know that hibernate can only access the database via a session, So If an entity is detached from the session and when we try to access an association (via a proxy or collection wrapper) that has not yet been loaded, Hibernate throws a LazyInitializationException.

The cure is to ensure either that the entity is made persistent again by attaching it to a session or that all of the fields that will be required are accessed (so they are loaded into entity) before the entity is detached from the session.

That's all for this simple, yet very important concept i.e. how to load lazy object in hibernate. This can be a question for beginners in hibernate fetching strategies interview questions.

Hibernate criteria queries examples

Hibernate provides three different ways to retrieve data from database. We have already discussed **HQL** and **native SQL** queries. Now we will discuss our third option i.e. hibernate criteria queries. The **criteria query API** lets you build nested, structured query expressions in Java, **providing a compile-time syntax checking that is not possible with a query language like HQL or SQL**.

The **Criteria API** also includes query by example (QBE) functionality. This lets you supply example objects that contain the properties you would like to retrieve instead of having to step-by-step spell out the components of the query. It also includes projection and aggregation methods, including count(). Let's explore it's different features in detail.

1. Hibernate criteria example

The Criteria API allows you to build up a criteria query object programmatically; the org.hibernate.Criteria interface defines the available methods for one of these objects. The Hibernate Session interface contains several overloaded createCriteria() methods.

Pass the persistent object's class or its entity name to the createCriteria() method, and hibernate will create a Criteria object that returns instances of the persistence object's class when your application executes a criteria query.

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

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

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

2. Hibernate criteria – using Restrictions

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

2.1. Restrictions.eq() Example

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

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

Above query will search all products having description as "Mouse".

2.2. Restrictions.ne() Example

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

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

Above query will search all products having description anything but not “Mouse”.
You cannot use the not-equal restriction to retrieve records with a NULL value in the database for that property (in SQL, and therefore in Hibernate, NULL represents the absence of data, and so cannot be compared with data). If you need to retrieve objects with NULL properties, you will have to use the isNull() restriction.

2.3. Restrictions.like() and Restrictions.ilike() example

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

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

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

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

2.4. Restrictions.isNull() and Restrictions.isNotNull() example

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

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

2.5. Restrictions.gt(), Restrictions.ge(), Restrictions.lt() and Restrictions.le() examples

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

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

2.6. Combining Two or More Criteria Examples

Moving on, we can start to do more complicated queries with the Criteria API. For example, we can

combine AND and OR restrictions in logical expressions. When we add more than one constraint to a criteria query, it is interpreted as an AND, like so:

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

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

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

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

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

2.7. Using Disjunction Objects with Criteria

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

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

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

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

2.8. Restrictions.sqlRestriction() Example

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

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

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

3. Hibernate criteria – paging through the result set

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

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

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

TO BE ADDED REMAINING LATER

Hibernate – How to Define Association Mappings between Entities

In previous tutorial, we learned about four persistent states of hibernate entities and there we noted that hibernate can identify java objects as persistent only when they are annotated with certain annotations; otherwise they are treated as simple plain java objects with no direct relation to database. Moving on, when we annotate the java classes with JPA annotations and make them persistent entities, we can face situations where two entities can be related and must be referenced from each other, in either uni-direction or in bi-direction. Let's understand few basic things before actually creating references between hibernate entities.

Understanding Entities and Their Associations

Entities can contain references to other entities, either directly as an embedded property or field, or indirectly via a collection of some sort (arrays, sets, lists, etc.). These associations are represented using foreign key relationships in the underlying tables. These foreign keys will rely on the identifiers used by participating tables.

When only one of the pair of entities contains a reference to the other, the association is unidirectional. If the association is mutual, then it is referred to as bidirectional.

A common mistake by beginners, when designing entity models, is to try to make all associations bidirectional. Remember that associations that are not a natural part of the object model should not be forced into it. Hibernate Query Language (HQL) often proves a more natural way to access the required information when needed.

Ideally, we would like to dictate that only changes to one end of the relationship will result in any updates to the foreign key; and indeed, Hibernate allows us to do this by marking one end of the association as being managed by the other.

In hibernate mapping associations, one (and only one) of the participating classes is referred to as "managing the relationship" and other one is called "managed by" using 'mappedBy' property. We should not make both ends of association "managing the relationship". Never do it.

Note that "mappedBy" is purely about how the foreign key relationships between entities are saved. It has nothing to do with saving the entities themselves using cascade functionality.

While Hibernate lets us specify that changes to one association will result in changes to the database, it does not allow us to cause changes to one end of the association to be automatically reflected in the other end in the Java POJOs. We must use cascading capabilities for doing so.

Understanding Associations with Example

Let's quickly build and example to understand what we have read above about associations about entities and how this should be done. I am using two simple entities (AccountEntity and EmployeeEntity) for this example and then I will create one-to-one association between them. Then we will see how various association options change the runtime behavior and complexity.

AccountEntity.java

```
@Entity
@Table(name = "Account")
public class AccountEntity implements Serializable
{
```

```
  private static final long serialVersionUID = 1L;

  @Id
  @Column(name = "ID", unique = true, nullable = false)
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  private Integer          accountId;

  @Column(name = "ACC_NO", unique = false, nullable = false, length = 100)
  private String           accountNumber;

  //We will define the association here
  EmployeeEntity           employee;

  //Getters and Setters are not shown for brevity
}
```

EmployeeEntity.java

```
@Entity
@Table(name = "Employee")
public class EmployeeEntity implements Serializable
{
  private static final long serialVersionUID = -1798070786993154676L;
  @Id
  @Column(name = "ID", unique = true, nullable = false)
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  private Integer          employeeId;
  @Column(name = "FIRST_NAME", unique = false, nullable = false, length = 100)
  private String           firstName;
  @Column(name = "LAST_NAME", unique = false, nullable = false, length = 100)
  private String           lastName;

  //We will define the association here
  AccountEntity            account;

  //Getters and Setters are not shown for brevity
}
```

Scenario 1) Association managed by both entities

In this type of association, we will define the association as below using @OneToOne annotation.

```
//Inside EmployeeEntity.java
@OneToOne
AccountEntity          account;

//Inside AccountEntity.java
@OneToOne
EmployeeEntity                 employee;
```

With above association, both ends are managing the association so both must be updated with informa-

tion of each other using setter methods defined in entity java files. If you fail to do so, you will not be able to fetch the associated entity information and it will be returned as null.

```
public class TestHibernate {
  public static void main(String[] args) {
    Session sessionOne = HibernateUtil.getSessionFactory().openSession();
    sessionOne.beginTransaction();

    // Create new Employee object
    EmployeeEntity emp = new EmployeeEntity();
    emp.setFirstName("Lokesh");
    emp.setLastName("Gupta");

    // Create new Employee object
    AccountEntity acc = new AccountEntity();
    acc.setAccountNumber("DUMMY_ACCOUNT");
    emp.setAccount(acc);
    //acc.setEmployee(emp);

    sessionOne.save(acc);
    sessionOne.save(emp);
    sessionOne.getTransaction().commit();

    Integer genEmpId = emp.getEmployeeId();
    Integer genAccId  = acc.getAccountId();

    Session sessionTwo = HibernateUtil.getSessionFactory().openSession();
    sessionTwo.beginTransaction();
    EmployeeEntity employee = (EmployeeEntity) sessionTwo.get(EmployeeEntity.class, genEmpId);
    AccountEntity account = (AccountEntity) sessionTwo.get(AccountEntity.class, genAccId);

    System.out.println(employee.getEmployeeId());
    System.out.println(employee.getAccount().getAccountNumber());
    System.out.println(account.getAccountId());
    System.out.println(account.getEmployee().getEmployeeId());

    HibernateUtil.shutdown();
  }
}
```

Output:

```
Hibernate: insert into Account (ACC_NO, employee_ID, ID) values (?, ?, ?)
Hibernate: insert into Employee (account_ID, FIRST_NAME, LAST_NAME, ID) values (?, ?, ?, ?)
Hibernate: select employeeen0_.ID as ID1_1_0_, employeeen0_.account_ID as account_4_1_0_, em-
ployeeen0_.FIRST_NAME as FIRST_NA2_1_0_,
                        employeeen0_.LAST_NAME as LAST_NAM3_1_0_, accountent1_.ID as
ID1_0_1_, accountent1_.ACC_NO as ACC_NO2_0_1_,
                        accountent1_.employee_ID as employee3_0_1_, employeeen2_.ID as ID1_1_2_,
employeeen2_.account_ID as account_4_1_2_,
```

```
                employeeeen2_.FIRST_NAME as FIRST_NA2_1_2_, employe-
een2_.LAST_NAME as LAST_NAM3_1_2_ from Employee
                employeeeen0_ left outer join Account accountent1_ on employeeeen0_.ac-
count_ID=accountent1_.ID
                left outer join Employee employeeeen2_ on accountent1_.employee_ID=employe-
een2_.ID where employeeeen0_.ID=?

20
DUMMY_ACCOUNT
10
Exception in thread "main" java.lang.NullPointerException
        at com.howtodoinjava.test.TestHibernate.main(TestHibernate.java:43)
```

You see that I had set the account entity in employee entity, so I am able to get it. BUT I commented the line “acc.setEmployee(emp);” and thus not set the employee entity inside account entity, so I am not able to get it.

Also observe the insert queries above. Both tables have foreign key association with column names employee_ID and account_ID respectively. It’s example of circular dependency in data and can easily bring down your application any time.

To correctly store above relationship, you must un-comment the line “acc.setEmployee(emp);“. After uncommenting the line, you will be able to store and retrieve the association as desired but still there is circular dependency.

After uncommenting the line, output will be like below:

```
Hibernate: insert into Account (ACC_NO, employee_ID, ID) values (?, ?, ?)
Hibernate: insert into Employee (account_ID, FIRST_NAME, LAST_NAME, ID) values (?, ?, ?, ?)
Hibernate: update Account set ACC_NO=?, employee_ID=? where ID=?
Hibernate: select employeeen0_.ID as ID1_1_0_, employeeen0_.account_ID as account_4_1_0_, em-
ployeeen0_.FIRST_NAME as FIRST_NA2_1_0_,
                employeeen0_.LAST_NAME as LAST_NAM3_1_0_, accountent1_.ID as
ID1_0_1_, accountent1_.ACC_NO as ACC_NO2_0_1_,
                accountent1_.employee_ID as employee3_0_1_, employeeen2_.ID as ID1_1_2_,
employeeen2_.account_ID as account_4_1_2_,
                employeeen2_.FIRST_NAME as FIRST_NA2_1_2_, employe-
een2_.LAST_NAME as LAST_NAM3_1_2_ from Employee
                employeeeen0_ left outer join Account accountent1_ on employeeeen0_.ac-
count_ID=accountent1_.ID
                left outer join Employee employeeen2_ on accountent1_.employee_ID=employe-
een2_.ID where employeeeen0_.ID=?

20
DUMMY_ACCOUNT
10
20
```

Scenario 2) Association managed by single entity

Let's say association is managed by EmployeeEntity then the annotations in both entity will look like below.

```
//Inside EmployeeEntity.java
@OneToOne
AccountEntity          account;

//Inside AccountEntity.java
@OneToOne (mappedBy = "account")
EmployeeEntity                 employee;
```

Now to tell hibernate the association is managed by EmployeeEntity, we will add 'mappedBy' attribute inside AccountEntity to make it manageable. Now to test above code we will have to set the association only once using **"emp.setAccount(acc);**" and employee entity is which is managing the relationship. AccountEntity dos not need to know anything explicitly.

Let's look at below test run.

```
Session sessionOne = HibernateUtil.getSessionFactory().openSession();
sessionOne.beginTransaction();

// Create new Employee object
EmployeeEntity emp = new EmployeeEntity();
emp.setFirstName("Lokesh");
emp.setLastName("Gupta");

// Create new Employee object
AccountEntity acc = new AccountEntity();
acc.setAccountNumber("DUMMY_ACCOUNT");
emp.setAccount(acc);
//acc.setEmployee(emp);

sessionOne.save(acc);
sessionOne.save(emp);
sessionOne.getTransaction().commit();

Integer genEmpId = emp.getEmployeeId();
Integer genAccId  = acc.getAccountId();

Session sessionTwo = HibernateUtil.getSessionFactory().openSession();
sessionTwo.beginTransaction();
EmployeeEntity employee = (EmployeeEntity) sessionTwo.get(EmployeeEntity.class, genEmpId);
AccountEntity account = (AccountEntity) sessionTwo.get(AccountEntity.class, genAccId);

System.out.println(employee.getEmployeeId());
System.out.println(employee.getAccount().getAccountNumber());
System.out.println(account.getAccountId());
System.out.println(account.getEmployee().getEmployeeId());
```

```
HibernateUtil.shutdown();
```

Output:

```
Hibernate: insert into Account (ACC_NO, ID) values (?, ?)
Hibernate: insert into Employee (account_ID, FIRST_NAME, LAST_NAME, ID) values (?, ?, ?, ?)
Hibernate: select employeeen0_.ID as ID1_1_0_, employeeen0_.account_ID as account_4_1_0_, em-
ployeeen0_.FIRST_NAME as FIRST_NA2_1_0_,
employeeen0_.LAST_NAME as LAST_NAM3_1_0_, accountent1_.ID as ID1_0_1_, accoun-
tent1_.ACC_NO as ACC_NO2_0_1_ from Employee employeeen0_
left outer join Account accountent1_ on employeeen0_.account_ID=accountent1_.ID where employe-
een0_.ID=?
Hibernate: select employeeen0_.ID as ID1_1_1_, employeeen0_.account_ID as account_4_1_1_, em-
ployeeen0_.FIRST_NAME as FIRST_NA2_1_1_,
employeeen0_.LAST_NAME as LAST_NAM3_1_1_, accountent1_.ID as ID1_0_0_, accoun-
tent1_.ACC_NO as ACC_NO2_0_0_ from Employee employeeen0_
left outer join Account accountent1_ on employeeen0_.account_ID=accountent1_.ID where employe-
een0_.account_ID=?

20
DUMMY_ACCOUNT
10
20
```

You see that you do not need to tell anything to account entity (‘acc.setEmployee(emp)‘ is commented). Employee entity is managing the association both ways.

Another observation is regarding foreign key columns which we have only one now i.e. account_ID is Employee table. So no circular dependency as well. All good.

Guide to Define Associations for Various Mappings

Above example shows how to manage association between entities in one-to-one mapping. In above example, we could have chosen the association managed by AccountEntity as well and things would have worked out pretty well with minor code changes. But in case of other mappings (e.g. One-to-many or Many-to-one), you will not have the liberty to define associations at your will. You need rules.

Below table shows how you can select the side of the relationship that should be made the owner of a bi-directional association. Remember that to make an association the owner, you must mark the other end as being mapped by the other.

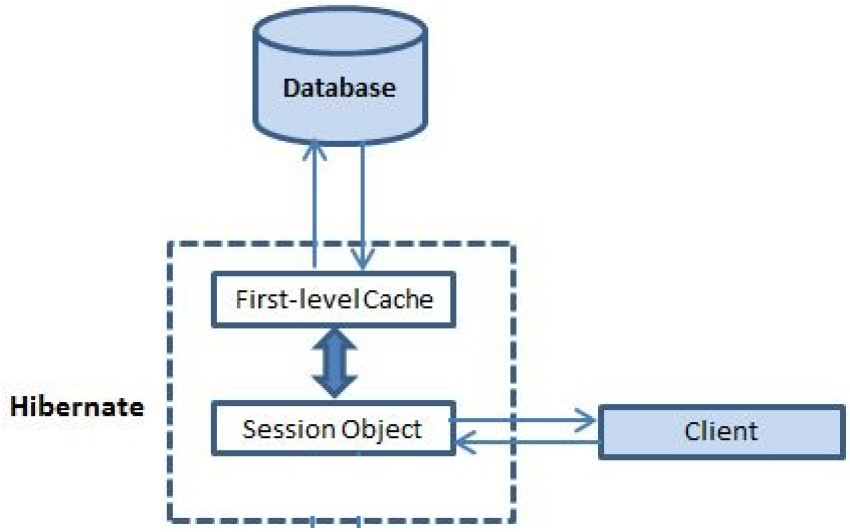
Type of Association	Options/ Usage
One-to-one	Either end can be made the owner, but one (and only one) of them should be; if you don’t specify this, you will end up with a circular dependency.
One-to-many	The many end must be made the owner of the association.
Many-to-one	This is the same as the one-to-many relationship viewed from the opposite perspective, so the same rule applies: the many end must be made the owner of the association.
Many-to-many	Either end of the association can be made the owner.

Understanding Hibernate First Level Cache with Example

Caching is a facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate achieves the second goal by implementing first level cache.

Fist level cache in hibernate is enabled by default and you do not need to do anything to get this functionality working. In fact, you can not disable it even forcefully.

Its easy to understand the first level cache if we understand the fact that **it is associated with Session object.** As we know session object is created on demand from session factory and it is lost, once the session is closed. Similarly, **first level cache associated with session object is available only till session object is live**. It is available to session object only and is not accessible to any other session object in any other part of application.



Important facts

- First level cache is associated with "session" object and other session objects in application can not see it.
- The scope of cache objects is of session. Once session is closed, cached objects are gone forever.
- First level cache is enabled by default and you can not disable it.
- When we query an entity first time, it is retrieved from database and stored in first level cache associated with hibernate session.
- If we query same object again with same session object, it will be loaded from cache and no sql query will be executed.
- The loaded entity can be removed from session using **evict()** method. The next loading of this entity will again make a database call if it has been removed using evict() method.
- The whole session cache can be removed using **clear()** method. It will remove all the entities stored in cache.

Lets verify above facts using examples.

First level cache retrieval example

In this example, I am retrieving DepartmentEntity object from database using hibernate session. I will retrieve it multiple times, and will observe the sql logs to see the differences.

```
//Open the hibernate session
Session session = HibernateUtil.getSessionFactory().openSession();
session.beginTransaction();

//fetch the department entity from database first time
DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
System.out.println(department.getName());

//fetch the department entity again
department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
System.out.println(department.getName());

session.getTransaction().commit();
HibernateUtil.shutdown();
```

Output:

```
Hibernate: select department0_.ID as ID0_0_, department0_.NAME as NAME0_0_ from DEPARTMENT department0_ where department0_.ID=?
Human Resource
Human Resource
```

As you can see that second "session.load()" statement does not execute select query again and load the department entity directly.

First level cache retrieval example with new session

With new session, entity is fetched from database again irrespective of it is already present in any other session in application.

```
//Open the hibernate session
Session session = HibernateUtil.getSessionFactory().openSession();
session.beginTransaction();

Session sessionTemp = HibernateUtil.getSessionFactory().openSession();
sessionTemp.beginTransaction();
try
{
        //fetch the department entity from database first time
        DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
        System.out.println(department.getName());

        //fetch the department entity again
        department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
        System.out.println(department.getName());

        department = (DepartmentEntity) sessionTemp.load(DepartmentEntity.class, new Inte-
```

```
ger(1));
        System.out.println(department.getName());
}
finally
{
        session.getTransaction().commit();
        HibernateUtil.shutdown();

        sessionTemp.getTransaction().commit();
        HibernateUtil.shutdown();
}
```

Output:

```
Hibernate: select department0_.ID as ID0_0_, department0_.NAME as NAME0_0_ from DEPART-
MENT department0_ where department0_.ID=?
Human Resource
Human Resource

Hibernate: select department0_.ID as ID0_0_, department0_.NAME as NAME0_0_ from DEPART-
MENT department0_ where department0_.ID=?
Human Resource
```

You can see that even if the department entity was stored in "session" object, still another database query was executed when we use another session object "sessionTemp".

Removing cache objects from first level cache example

Though we can not disable the first level cache in hibernate, but we can certainly remove some of objects from it when needed. This is done using two methods :

- **evict()**
- **clear()**

Here evict() is used to remove a particular object from cache associated with session, and clear() method is used to remove all cached objects associated with session. So they are essentially like remove one and remove all.

```
//Open the hibernate session
Session session = HibernateUtil.getSessionFactory().openSession();
session.beginTransaction();
try
{
        //fetch the department entity from database first time
        DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new
Integer(1));
        System.out.println(department.getName());

        //fetch the department entity again
        department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
        System.out.println(department.getName());
```

```
        session.evict(department);
        //session.clear();

        department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
        System.out.println(department.getName());
}
finally
{
        session.getTransaction().commit();
        HibernateUtil.shutdown();
}
```

Output:
Hibernate: select department0_.ID as ID0_0_, department0_.NAME as NAME0_0_ from DEPARTMENT department0_ where department0_.ID=?
Human Resource
Human Resource

Hibernate: select department0_.ID as ID0_0_, department0_.NAME as NAME0_0_ from DEPARTMENT department0_ where department0_.ID=?
Human Resource

Clearly, evict() method removed the department object from cache so that it was fetched again from database.

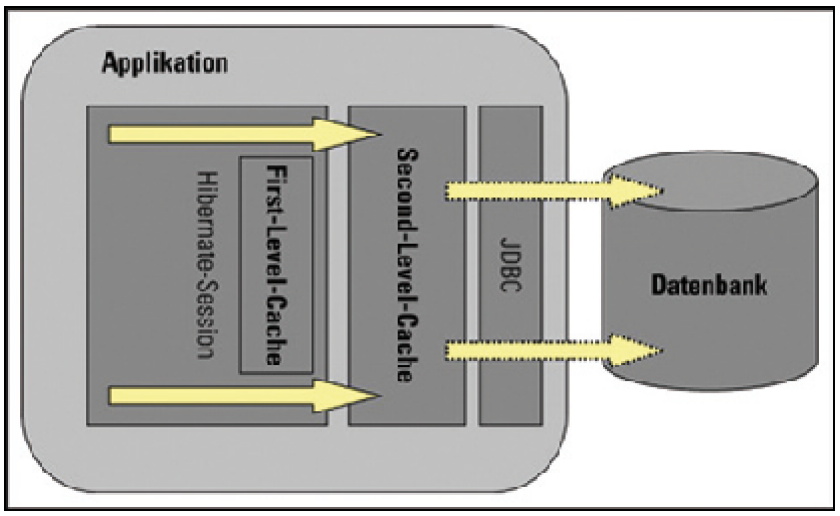
How Hibernate Second Level Cache Works?

Caching is facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate also provide this caching functionality, in two layers.

- Fist level cache: This is enabled by default and works in session scope. Read more about hibernate first level cache.
- Second level cache: This is apart from first level cache which is available to be used globally in session factory scope.

Above statement means, second level cache is created in session factory scope and is available to be used in all sessions which are created using that particular session factory.
It also means that once session factory is closed, all cache associated with it die and cache manager also closed down.

Further, It also means that if you have two instances of session factory (normally no application does that), you will have two cache managers in your application and while accessing cache stored in physical store, you might get unpredictable results like cache-miss.



How second level cache works

Lets write all the facts point by point:

- Whenever hibernate session try to load an entity, the very first place it look for cached copy of entity in first level cache (associated with particular hibernate session).
- If cached copy of entity is present in first level cache, it is returned as result of load method.
- If there is no cached entity in first level cache, then second level cache is looked up for cached entity.
- If second level cache has cached entity, it is returned as result of load method. But, before returning the entity, it is stored in first level cache also so that next invocation to load method for entity will return the entity from first level cache itself, and there will not be need to go to second level cache again.
- If entity is not found in first level cache and second level cache also, then database query is executed and entity is stored in both cache levels, before returning as response of load() method.
- Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.
- If some user or process make changes directly in database, the there is no way that second level cache update itself until “timeToLiveSeconds” duration has passed for that cache region. In this case, it is good idea to invalidate whole cache and let hibernate build its cache once again. You can use below code snippet to invalidate whole hibernate second level cache.

```
**
 * Evicts all second level cache hibernate entites. This is generally only
 * needed when an external application modifies the databaase.
 */
public void evict2ndLevelCache() {
   try {
      Map<String, ClassMetadata> classesMetadata = sessionFactory.getAllClassMetadata();
      for (String entityName : classesMetadata.keySet()) {
         logger.info("Evicting Entity from 2nd level cache: " + entityName);
         sessionFactory.evictEntity(entityName);
      }
   } catch (Exception e) {
```

```
        logger.logp(Level.SEVERE, "SessionController", "evict2ndLevelCache", "Error evicting 2nd level hibernate cache entities: ", e);
    }
}
```

To understand more using examples, I wrote an application for testing in which I configured EhCache as 2nd level cache. Lets see various scenarios:

a) Entity is fetched very first time

```
DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
System.out.println(department.getName());

System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount());
//Prints 1
System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount());
//Prints 0
```

Output: 1 0

Explanation: Entity is not present in either 1st or 2nd level cache so, it is fetched from database.

b) Entity is fetched second time

```
//Entity is fecthed very first time
DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
System.out.println(department.getName());

//fetch the department entity again
department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
System.out.println(department.getName());

System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount());
//Prints 1
System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount());
//Prints 0
```

Output: 1 0

Explanation: Entity is present in first level cache so, it is fetched from there. No need to go to second level cache.

c) Entity is evicted from first level cache and fetched again

```
//Entity is fecthed very first time
DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
System.out.println(department.getName());

//fetch the department entity again
```

```
department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
System.out.println(department.getName());

//Evict from first level cache
session.evict(department);

department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
System.out.println(department.getName());

System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount());
//Prints 1
System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount());
//Prints 1
```

Output: 1 1

Explanation: First time entity is fetched from database. Which cause it store in 1st and 2nd level cache. Second load call fetched from first level cache. Then we evicted entity from 1st level cache. So third load() call goes to second level cache and getSecondLevelCacheHitCount() returns 1.

d) Access second level cache from another session

```
//Entity is fecthed very first time
DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
System.out.println(department.getName());

//fetch the department entity again
department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
System.out.println(department.getName());

//Evict from first level cache
session.evict(department);

department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
System.out.println(department.getName());

department = (DepartmentEntity) anotherSession.load(DepartmentEntity.class, new Integer(1));
System.out.println(department.getName());

System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntityFetchCount());
//Prints 1
System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecondLevelCacheHitCount());
//Prints 2
```

Output: 1 2

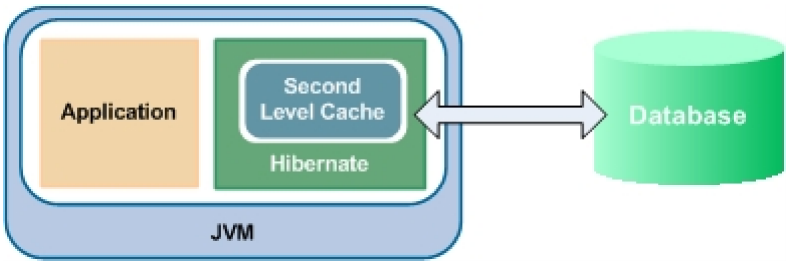
Explanation: When another session created from same session factory try to get entity, it is successfully looked up in second level cache and no database call is made.

Hibernate EhCache Configuration Tutorial

Caching is facility provided by ORM frameworks which help users to get fast running web application, while help framework itself to reduce number of queries made to database in a single transaction. Hibernate also provide this caching functionality, in two layers.

- Fist level cache: This is enabled by default and works in session scope. Read more about hibernate first level cache.
- Second level cache: This is apart from first level cache which is available to be used globally in session factory scope.

In this tutorial, I am giving an example using ehcache configuration as second level cache in hibernate.



How second level cache works

Lets write all the facts point by point:

- Whenever hibernate session try to load an entity, the very first place it look for cached copy of entity in first level cache (associated with particular hibernate session).
- If cached copy of entity is present in first level cache, it is returned as result of load method.
- If there is no cached entity in first level cache, then second level cache is looked up for cached entity.
- If second level cache has cached entity, it is returned as result of load method. But, before returning the entity, it is stored in first level cache also so that next invocation to load method for entity will return the entity from first level cache itself, and there will not be need to go to second level cache again.
- If entity is not found in first level cache and second level cache also, then database query is executed and entity is stored in both cache levels, before returning as response of load() method.
- Second level cache validate itself for modified entities, if modification has been done through hibernate session APIs.
- If some user or process make changes directly in database, the there is no way that second level cache update itself until “timeToLiveSeconds” duration has passed for that cache region. In this case, it is good idea to invalidate whole cache and let hibernate build its cache once again. You can use below code snippet to invalidate whole hibernate second level cache.

About EhCache

Terracotta Ehcache is a popular open source Java cache that can be used as a Hibernate second level cache. It can be used as a standalone second level cache, or can be configured for clustering to provide a replicated coherent second level cache.

The maven dependency is for Ehcache 2.0 and any upgrades is:

```
<dependency>
        <groupId>net.sf.ehcache</groupId>
        <artifactId>ehcache</artifactId>
        <version>[2.0.0]</version>
        <type>pom</type>
</dependency>
```

Configuring EhCache
To configure ehcache, you need to do two steps:

1. configure Hibernate for second level caching
2. specify the second level cache provider

Hibernate 4.x and above

```
<property key="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.region.factory_class">org.hibernate.cache.ehcache.EhCacheRegion-
Factory</property>
```

Hibernate 3.3 and above

```
<property key="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.region.factory_class">net.sf.ehcache.hibernate.EhCacheRegionFac-
tory</property>
```

Hibernate 3.2 and below

```
<property key="hibernate.cache.use_second_level_cache">true</property>
<property name="hibernate.cache.region.provider_class">net.sf.ehcache.hibernate.EhCacheProvi-
der</property>
```

Configuring entity objects

This may done in two ways.
1) If you are using hbm.xml files then use below configuration:

```
<class name="com.application.entity.DepartmentEntity" table="...">
   <cache usage="read-write"/>
</class>
```

2) Otherwise, if you are using annotations, use these annotations:

```
@Entity
@Cache(usage=CacheConcurrencyStrategy.READ_ONLY,  region="department")
public class DepartmentEntity implements Serializable
{
        //code
}
```

For both options, caching strategy can be of following types:

1. **none** : No caching will happen.
2. **read-only** : If your application needs to read, but not modify, instances of a persistent class, a read-only cache can be used.
3. **read-write** : If the application needs to update data, a read-write cache might be appropriate.
4. **nonstrict-read-write** : If the application only occasionally needs to update data (i.e. if it is extremely unlikely that two transactions would try to update the same item simultaneously), and strict

transaction isolation is not required, a nonstrict-read-write cache might be appropriate.

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

Query caching

You can also enable query caching. To do so configure it in your hbm.xml:

<property key="hibernate.cache.use_query_cache">true</property>

and where queries are defined in your code, add the method call setCacheable(true) to the queries that should be cached:

sessionFactory.getCurrentSession().createQuery("...").setCacheable(true).list();

By default, Ehcache will create separate cache regions for each entity that you configure for caching. You can change the defaults for these regions by adding the configuration to your ehcache.xml. To provide this configuration file, use this property in hibernate configuration:

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

And use below configuration to override the default configuration:

```
<cache
   name="com.somecompany.someproject.domain.Country"
   maxElementsInMemory="10000"
   eternal="false"
   timeToIdleSeconds="300"
   timeToLiveSeconds="600"
   overflowToDisk="true"
/>
```

Please note that in ehcache.xml, if **eternal=”true”** then we should not write timeToIdealSeconds, timeToLiveSeconds, hibernate will take care about those values
So if you want to give values manually better use eternal=”false” always, so that we can assign values into timeToIdealSeconds, timeToLiveSeconds manually.

timeToIdealSeconds=”seconds” means, if the object in the global cache is ideal, means not using by any other class or object then it will be waited for some time we specified and deleted from the global cache if time is exceeds more than timeToIdealSeconds value.

timeToLiveSeconds=”seconds” means, the other Session or class using this object or not, i mean may be it is using by other sessions or may not, what ever the situation might be, once it competed the time specified timeToLiveSeconds, then it will be removed from the global cache by hibernate.

Example application
In our example application, I have one DepartmentEntity for which I want to enable second level cache using ehcache. Lets record the changes step by step:

1) hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
      <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernatedemo</property>
      <property name="hibernate.connection.password">password</property>
      <property name="hibernate.connection.username">root</property>
      <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
      <property name="show_sql">true</property>
                <property name="hbm2ddl.auto">create</property>
                <property  name="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</property>
      <mapping class="hibernate.test.dto.DepartmentEntity"></mapping>
   </session-factory>
</hibernate-configuration>
```

2) DepartmentEntity.java

```
package hibernate.test.dto;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;

import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity (name = "dept")
@Table(name = "DEPARTMENT", uniqueConstraints = {
                @UniqueConstraint(columnNames = "ID"),
                @UniqueConstraint(columnNames = "NAME") })

@Cache(usage=CacheConcurrencyStrategy.READ_ONLY, region="department")

public class DepartmentEntity implements Serializable {
```

```
	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "ID", unique = true, nullable = false)
	private Integer id;

	@Column(name = "NAME", unique = true, nullable = false, length = 100)
	private String name;

	public Integer getId() {
		return id;
	}

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

	public String getName() {
		return name;
	}

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

3) HibernateUtil.java

```
package hibernate.test;

import java.io.File;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

public class HibernateUtil  {
	private static final SessionFactory sessionFactory = buildSessionFactory();

   private static SessionFactory buildSessionFactory()  {
      try
      {
         // Create the SessionFactory from hibernate.cfg.xml
           return new AnnotationConfiguration().configure(new File("hibernate.cgf.xml")).buildSession-
Factory();
      }
      catch (Throwable ex) {
         // Make sure you log the exception, as it might be swallowed
         System.err.println("Initial SessionFactory creation failed." + ex);
```

```
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
            // Close caches and connection pools
            getSessionFactory().close();
    }
}
```

4) TestHibernateEhcache.java

```
public class TestHibernateEhcache
{
        public static void main(String[] args)
        {
                storeData();

                try
                {
                        //Open the hibernate session
                        Session session = HibernateUtil.getSessionFactory().openSession();
                        session.beginTransaction();

                        //fetch the department entity from database first time
                        DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
                        System.out.println(department.getName());

                        //fetch the department entity again; Fetched from first level cache
                        department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
                        System.out.println(department.getName());

                        //Let's close the session
                        session.getTransaction().commit();
                        session.close();

                        //Try to get department in new session
                        Session anotherSession = HibernateUtil.getSessionFactory().openSession();
                        anotherSession.beginTransaction();

                        //Here entity is already in second level cache so no database query will be hit
                        department = (DepartmentEntity) anotherSession.load(DepartmentEntity.class, new Integer(1));
                        System.out.println(department.getName());
```

```
                anotherSession.getTransaction().commit();
                anotherSession.close();
            }
            finally
            {
                System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntity-
FetchCount()); //Prints 1
                System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecon-
dLevelCacheHitCount()); //Prints 1

                HibernateUtil.shutdown();
            }
        }

        private static void storeData()
        {
            Session session = HibernateUtil.getSessionFactory().openSession();
            session.beginTransaction();

            DepartmentEntity department = new DepartmentEntity();
            department.setName("Human Resource");

            session.save(department);
            session.getTransaction().commit();
        }
}
```

Output:

```
Hibernate: insert into DEPARTMENT (NAME) values (?)
Hibernate: select department0_.ID as ID0_0_, department0_.NAME as NAME0_0_ from DEPART-
MENT department0_ where department0_.ID=?
Human Resource
Human Resource
Human Resource
1
1
```

In above output, first time the department is fetched from database. but next two times it is fetched from cache. Last fetch is from second level cache.

Example application
In our example application, I have one DepartmentEntity for which I want to enable second level cache using ehcache. Lets record the changes step by step:

1) hibernate.cfg.xml

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
"-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
   <session-factory>
      <property name="hibernate.connection.driver_class">com.mysql.jdbc.Driver</property>
         <property name="hibernate.connection.url">jdbc:mysql://localhost:3306/hibernatedemo</property>
      <property name="hibernate.connection.password">password</property>
      <property name="hibernate.connection.username">root</property>
      <property name="hibernate.dialect">org.hibernate.dialect.MySQLDialect</property>
      <property name="show_sql">true</property>
                  <property name="hbm2ddl.auto">create</property>
                  <property    name="hibernate.cache.provider_class">org.hibernate.cache.EhCacheProvider</property>
      <mapping class="hibernate.test.dto.DepartmentEntity"></mapping>
   </session-factory>
</hibernate-configuration>
```

2) DepartmentEntity.java

```
package hibernate.test.dto;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;
import javax.persistence.UniqueConstraint;

import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;

@Entity (name = "dept")
@Table(name = "DEPARTMENT", uniqueConstraints = {
                  @UniqueConstraint(columnNames = "ID"),
                  @UniqueConstraint(columnNames = "NAME") })

@Cache(usage=CacheConcurrencyStrategy.READ_ONLY, region="department")
```

```
public class DepartmentEntity implements Serializable {

	private static final long serialVersionUID = 1L;

	@Id
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	@Column(name = "ID", unique = true, nullable = false)
	private Integer id;

	@Column(name = "NAME", unique = true, nullable = false, length = 100)
	private String name;

	public Integer getId() {
		return id;
	}

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

	public String getName() {
		return name;
	}

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

3) HibernateUtil.java

```
package hibernate.test;

import java.io.File;

import org.hibernate.SessionFactory;
import org.hibernate.cfg.AnnotationConfiguration;

public class HibernateUtil
{
	private static final SessionFactory sessionFactory = buildSessionFactory();

   private static SessionFactory buildSessionFactory()
   {
      try
      {
         // Create the SessionFactory from hibernate.cfg.xml
           return new AnnotationConfiguration().configure(new File("hibernate.cgf.xml")).buildSession-
```

```
Factory();
        }
        catch (Throwable ex) {
            // Make sure you log the exception, as it might be swallowed
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
            // Close caches and connection pools
            getSessionFactory().close();
    }
}
```

4) TestHibernateEhcache.java

```
public class TestHibernateEhcache
{
        public static void main(String[] args)
        {
                storeData();

                try
                {
                        //Open the hibernate session
                        Session session = HibernateUtil.getSessionFactory().openSession();
                        session.beginTransaction();

                        //fetch the department entity from database first time
                        DepartmentEntity department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
                        System.out.println(department.getName());

                        //fetch the department entity again; Fetched from first level cache
                        department = (DepartmentEntity) session.load(DepartmentEntity.class, new Integer(1));
                        System.out.println(department.getName());

                        //Let's close the session
                        session.getTransaction().commit();
                        session.close();

                        //Try to get department in new session
                        Session anotherSession = HibernateUtil.getSessionFactory().openSession();
```

```
                anotherSession.beginTransaction();

                //Here entity is already in second level cache so no database query will be hit
                department = (DepartmentEntity) anotherSession.load(DepartmentEntity.class,
new Integer(1));
                System.out.println(department.getName());

                anotherSession.getTransaction().commit();
                anotherSession.close();
            }
            finally
            {
                System.out.println(HibernateUtil.getSessionFactory().getStatistics().getEntity-
FetchCount()); //Prints 1
                System.out.println(HibernateUtil.getSessionFactory().getStatistics().getSecon-
dLevelCacheHitCount()); //Prints 1

                HibernateUtil.shutdown();
            }
        }

        private static void storeData()
        {
            Session session = HibernateUtil.getSessionFactory().openSession();
            session.beginTransaction();

            DepartmentEntity department = new DepartmentEntity();
            department.setName("Human Resource");

            session.save(department);
            session.getTransaction().commit();
        }
}
```

Output:

```
Hibernate: insert into DEPARTMENT (NAME) values (?)
Hibernate: select department0_.ID as ID0_0_, department0_.NAME as NAME0_0_ from DEPART-
MENT department0_ where department0_.ID=?
Human Resource
Human Resource
Human Resource
1
1
```

In above output, first time the department is fetched from database. but next two times it is fetched from cache. Last fetch is from second level cache.

Hibernate / JPA 2 Persistence Annotations Tutorial

This Hibernate (or JPA 2) Persistence Annotations Tutorial contains overview of all important annotations which you may need while annotating your java POJOs to make them act as persistent JPA entities. This tutorial first defines a POJO "EmployeeEntity", define some attribute inside it and also has respective getter and setter methods. As we learn the new annotations, we will apply these annotations on this EmployeeEntity and then we will understand what that specific annotation means.

Let's quickly list down the annotations, we are going to discuss in this tutorial.

Table of Contents

Primary Keys with @Id and @GeneratedValue

Each entity bean has to have a primary key, which you annotate on the class with the @Id annotation.

Typically, the primary key will be a single field, though it can also be a composite of multiple fields which we will see in later sections.

The placement of the @Id annotation determines the default access strategy that Hibernate will use for

the mapping. If the annotation is applied to a field as shown below, then “field access” will be used.

```
@Id
private Integer employeeId;
```

If, instead, the annotation is applied to the accessor for the field then property access will be used.

```
@Id
public Integer getEmployeeId()
{
 return employeeId;
}
```

Property access means that Hibernate will call the mutator/setter instead of actually setting the field directly, what it does in case of field access. This gives flexibility to alter the value of actual value set in id field if needed. Additionally, you can apply extra logic on setting of ‘id’ field in mutator for other fields as well.

By default, the @Id annotation will not create a primary key generation strategy, which means that you, as the code’s author, need to determine what valid primary keys are, by setting them explicitly calling setter methods. OR you can use @GeneratedValue annotation.

@GeneratedValue annotation takes a pair of attributes: **strategy** and **generator** as below:

```
@Id
@GeneratedValue (strategy = GenerationType.SEQUENCE)
private Integer employeeId;

//OR a more complex use can be

@Id
@GeneratedValue(strategy=GenerationType.TABLE , generator="employee_generator")
@TableGenerator(name="employee_generator",
                table="pk_table",
                pkColumnName="name",
                valueColumnName="value",
                allocationSize=100)
private Integer employeeId;
```

The strategy attribute must be a value from the javax.persistence.GeneratorType enumeration. If you do not specify a generator type, the default is AUTO. There are four different types of primary key generators on GeneratorType, as follows:

- **AUTO**: Hibernate decides which generator type to use, based on the database’s support for primary key generation.
- **IDENTITY**: The database is responsible for determining and assigning the next primary key.
- **SEQUENCE**: Some databases support a SEQUENCE column type. It uses @SequenceGenerator.
- **TABLE**: This type keeps a separate table with the primary key values. It uses @TableGenerator.

The generator attribute allows the use of a custom generation mechanism shown in above code example

Generating Primary Key Values with @SequenceGenerator

A sequence is a database object that can be used as a source of primary key values. It is similar to the use of an identity column type, except that a sequence is independent of any particular table and can therefore be used by multiple tables.

To declare the specific sequence object to use and its properties, you must include the @SequenceGenerator annotation on the annotated field. Here's an example:

```
@Id
@SequenceGenerator(name="seq1",sequenceName="HIB_SEQ")
@GeneratedValue(strategy=SEQUENCE,generator="seq1")
private Integer employeeId;
```

Here, a sequence-generation annotation named seq1 has been declared. This refers to the database sequence object called HIB_SEQ. The name seq1 is then referenced as the generator attribute of the @GeneratedValue annotation. Only the sequence generator name is mandatory; the other attributes will take sensible default values, but you should provide an explicit value for the sequenceName attribute as a matter of good practice anyway. If not specified, the sequenceName value to be used is selected by the persistence provider.

@Immutable and @NaturalId – Hibernate-Specific Annotations

In previous post, we learned about most commonly used JPA annotations in hibernate and we also learned there about the hibernate annotations which complement those JPA annotations. Using JPA annotations makes your application code portable to other JPA implementations which is a good thing to have. Apart from JPA annotations, hibernate have some of it's own annotations which you can use to have certain features in application code. But you must remember that it may make hard to make your code portable in future date.

1) @Immutable Annotation

The @Immutable annotation marks an entity as being, well, immutable. This is useful for situations in which your entity represents reference data–things like lists of states, genders, or other rarely mutated data.

Since things like states tend to be rarely changed, someone usually updates the data manually, via SQL or an administration application. Hibernate can cache this data aggressively, which needs to be taken into consideration; if the reference data changes, you'd want to make sure that the applications using it are notified (may use refresh() method) or restarted somehow.

What @Immutable annotation tells Hibernate is that any updates to an immutable entity should-not be passed on to the database without giving any error. @Immutable can be placed on a collection too; in this case, changes to the collection (additions, or removals) will cause a HibernateException to be thrown.

EmployeeEntity.java

```
import org.hibernate.annotations.Immutable;
```

```
@Immutable
@Entity
@Table(name = "Employee")
public class EmployeeEntity implements Serializable
{
 private static final long serialVersionUID = -1798070786993154676L;

 @Id
 @Column(name = "ID", unique = true, nullable = false)
 private Integer          employeeId;
 @Column(name = "FIRST_NAME", unique = false, nullable = false, length = 100)
 private String           firstName;
 @Column(name = "LAST_NAME", unique = false, nullable = false, length = 100)
 private String           lastName;

 //Setters and Getters
}
```

ImmutableAnnotationExample.java

```
public class ImmutableAnnotationExample
{
  public static void main(String[] args)
  {
    setupTestData();

    Session sessionOne = HibernateUtil.getSessionFactory().openSession();
    sessionOne.beginTransaction();

    //Load the employee in another session
    EmployeeEntity employee = (EmployeeEntity) sessionOne.load(EmployeeEntity.class, 1);

    //Update the first name
    employee.setFirstName("Alex");
    sessionOne.flush();
    sessionOne.close();

    Session sessionTwo = HibernateUtil.getSessionFactory().openSession();
    sessionTwo.beginTransaction();

    //Load the employee in another session
    EmployeeEntity employeeUpdated = (EmployeeEntity) sessionTwo.load(EmployeeEntity.class, 1);

    //Verify the first name
    System.out.println(employeeUpdated.getFirstName());

    sessionTwo.flush();
    sessionTwo.close();
    HibernateUtil.shutdown();
```

```
  }

  private static void setupTestData(){
    Session session = HibernateUtil.getSessionFactory().openSession();
    session.beginTransaction();

    //Create Employee
    EmployeeEntity emp = new EmployeeEntity();
    emp.setEmployeeId(1);
    emp.setFirstName("Lokesh");
    emp.setLastName("Gupta");
    session.save(emp);

    session.getTransaction().commit();
    session.close();
  }
}
```

Output:

```
Hibernate: insert into Employee (FIRST_NAME, LAST_NAME, ID) values (?, ?, ?)

Hibernate: select employeeen0_.ID as ID1_1_0_, employeeen0_.FIRST_NAME as FIRST_NA2_1_0_,
 employeeen0_.LAST_NAME as LAST_NAM3_1_0_
from Employee employeeen0_ where employeeen0_.ID=?

Hibernate: select employeeen0_.ID as ID1_1_0_, employeeen0_.FIRST_NAME as FIRST_NA2_1_0_,
 employeeen0_.LAST_NAME as LAST_NAM3_1_0_
from Employee employeeen0_ where employeeen0_.ID=?

Lokesh //Value didn't updated in database i.e. immutable
```

2) @NaturalId Annotation

We learned so much things in past tutorials about @Id annotation with @GeneratedValue to create primary keys for records in database. In most real life applications, these primary keys are “artificial primary keys” and referred only inside application runtime. However, there’s also the concept of a “natural ID“, which provides another convenient and logical way to refer to an entity, apart from an artificial or composite primary key.

An example of natural id might be a Social Security number or a Tax Identification Number in the United States, and PAN number in India. An entity (being a person or a corporation) might have an artificial primary key generated by Hibernate, but it also might have a unique tax identifier. Hibernate allows you to search and load entities based on these natural ids as well.

For natural IDs, there are two forms of load mechanisms; one uses the simple natural ID (where the natural ID is one and only one field), and the other uses named attributes as part of a composite natural ID.

Simple Natural Id Example

```
@Entity
```

```
@Table(name = "Employee")
public class EmployeeEntity implements Serializable
{
  private static final long serialVersionUID = -1798070786993154676L;
  @Id
  @Column(name = "ID", unique = true, nullable = false)
  private Integer          employeeId;
  @Column(name = "FIRST_NAME", unique = false, nullable = false, length = 100)
  private String           firstName;
  @Column(name = "LAST_NAME", unique = false, nullable = false, length = 100)
  private String           lastName;

  //Natural id can be SSN as well
  @NaturalId
  Integer SSN;

  //Setters and Getters
}
```

SimpleNaturalIdExample.java

```
public class SimpleNaturalIdExample
{
  public static void main(String[] args)
  {
    setupTestData();

    Session sessionOne = HibernateUtil.getSessionFactory().openSession();
    sessionOne.beginTransaction();

    //Load the employee
    EmployeeEntity employee1 = (EmployeeEntity) sessionOne.load(EmployeeEntity.class, 1);
    //Just to ensure that employee is loasded from DB
    System.out.println(employee1.getFirstName());

     //Get the employee for natural id i.e. SSN; This does not execute another SQL SELECT as entity is
 already present in session
        EmployeeEntity employee2 = (EmployeeEntity) sessionOne.bySimpleNaturalId(EmployeeEn-
 tity.class).load(12345);

    //Verify that employee1 and employee2 refer to same object
    assert(employee1 == employee2);

    sessionOne.flush();
    sessionOne.close();

    System.out.println("====================================");
    Session sessionTwo = HibernateUtil.getSessionFactory().openSession();
    sessionTwo.beginTransaction();
```

```
    //Get the employee for natural id i.e. SSN; entity is not present in this session
        EmployeeEntity employee = (EmployeeEntity) sessionTwo.bySimpleNaturalId(EmployeeEntity.class).load(12345);

    sessionTwo.flush();
    sessionTwo.close();
    HibernateUtil.shutdown();
  }

  private static void setupTestData(){
    Session session = HibernateUtil.getSessionFactory().openSession();
    session.beginTransaction();

    //Create Employee
    EmployeeEntity emp = new EmployeeEntity();
    emp.setEmployeeId(1);
    emp.setFirstName("Lokesh");
    emp.setLastName("Gupta");
    emp.setSSN(12345);
    session.save(emp);

    session.getTransaction().commit();
    session.close();
  }
}
```

Output:

```
Hibernate: insert into Employee (SSN, FIRST_NAME, LAST_NAME, ID) values (?, ?, ?, ?)

Hibernate: select employeeen0_.ID as ID1_1_0_, employeeen0_.SSN as SSN2_1_0_, employeeen0_.FIRST_NAME as FIRST_NA3_1_0_,
employeeen0_.LAST_NAME as LAST_NAM4_1_0_ from Employee employeeen0_ where employeeen0_.ID=?

Lokesh
==========
Hibernate: select employeeen_.ID as ID1_1_ from Employee employeeen_ where employeeen_.SSN=?

Hibernate: select employeeen0_.ID as ID1_1_0_, employeeen0_.SSN as SSN2_1_0_, employeeen0_.FIRST_NAME as FIRST_NA3_1_0_,
employeeen0_.LAST_NAME as LAST_NAM4_1_0_ from Employee employeeen0_ where employeeen0_.ID=?
```

Please watch closely that in case of entity already not present in session, and if you get entity using it's natural id then first primary id is fetched using natural id; and then entity is fetched using this primary id. If entity is already present in session, then reference of same entity is returned without executing

additional SELECT statements in database.

Composite Natural Id Example

```
@Entity
@Table(name = "Employee")
public class EmployeeEntity implements Serializable
{
  private static final long serialVersionUID = -1798070786993154676L;
  @Id
  @Column(name = "ID", unique = true, nullable = false)
  private Integer          employeeId;
  @Column(name = "FIRST_NAME", unique = false, nullable = false, length = 100)
  private String           firstName;
  @Column(name = "LAST_NAME", unique = false, nullable = false, length = 100)
  private String           lastName;

  //Natural id part 1
  @NaturalId
  Integer seatNumber;

  //Natural id part 2
  @NaturalId
  String departmentName;

  //Setters and Getters
}
```

CompositeNaturalIdExample.java

```
public class CompositeNaturalIdExample
{
  public static void main(String[] args)
  {
    setupTestData();

    Session sessionOne = HibernateUtil.getSessionFactory().openSession();
    sessionOne.beginTransaction();

    //Get the employee for natural id i.e. SSN; entity is not present in this session
    EmployeeEntity employee = (EmployeeEntity) sessionOne.byNaturalId(EmployeeEntity.class)
                                        .using("seatNumber", 12345)
                                        .using("departmentName", "IT")
                                        .load();

    System.out.println(employee.getFirstName());

    sessionOne.flush();
```

```
    sessionOne.close();
    HibernateUtil.shutdown();
  }

  private static void setupTestData(){
    Session session = HibernateUtil.getSessionFactory().openSession();
    session.beginTransaction();

    //Create Employee
    EmployeeEntity emp = new EmployeeEntity();
    emp.setEmployeeId(1);
    emp.setFirstName("Lokesh");
    emp.setLastName("Gupta");
    emp.setSeatNumber(12345);
    emp.setDepartmentName("IT");
    session.save(emp);

    session.getTransaction().commit();
    session.close();
  }
}
```

Output:

```
Hibernate: insert into Employee (departmentName, FIRST_NAME, LAST_NAME, seatNumber, ID)
 values (?, ?, ?, ?, ?)

Hibernate: select employeeen_.ID as ID1_1_ from Employee employeeen_ where employeeen_.depart-
 mentName=? and employeeen_.seatNumber=?

Hibernate: select employeeen0_.ID as ID1_1_0_, employeeen0_.departmentName as departme2_1_0_,
 employeeen0_.FIRST_NAME as FIRST_NA3_1_0_,
employeeen0_.LAST_NAME as LAST_NAM4_1_0_, employeeen0_.seatNumber as seatNumb5_1_0_
 from Employee employeeen0_ where employeeen0_.ID=?

Lokesh
```

Entity fetching logic for composite natural id is same as simple natural id. No difference apart from use of multiple natural keys instead one