Hibernate:

# Q:[Differences between session vs session factory - Hibernate?](https://stackoverflow.com/questions/22470968/differences-between-session-vs-session-factory-hibernate)

SessionFactory is Hibernate’s concept of a single datastore and is threadsafe so that many threads can access it concurrently and request for sessions and immutable cache of compiled mappings for a single database.

No, Session is not Thread Safe. A Session is a light weight and a non-threadsafe object (No, you cannot share it between threads) that represents a single unit-of-work with the database. Sessions are opened by a SessionFactory and then are closed when all work is complete. Session is the primary interface for the persistence service. A session obtains a database connection lazily (i.e. only when required)

Q: Is Hibernate's session thread safe?

t is not intended that implementors be threadsafe. Instead each thread/transaction should obtain its own instance from a SessionFactory.

Even with this in mind, your behaviour might still not be what you expect, because transactions come into play. You will have to set a proper [transaction isolation level](http://en.wikipedia.org/wiki/Isolation_(database_systems)). See the [configuration guide](http://docs.jboss.org/hibernate/core/3.3/reference/en/html/session-configuration.html), hibernate.connection.isolation property.

It depends on how you are creating a session.

Session can be created in two ways in hibernate.

1. getCurrentSession()

Yes. It offers thread safety as it'll ensure that it'll create a session for each thread if session not exist. transaction and automatic session closing is attached to this.

1. openSession()

It's not thread safe. developer manually needs to manage transactions and session flush and close operations.

Q: What is the difference between session.commit() and session.flush()?

Here are some relevant quotes from the [documentation](http://www.sqlalchemy.org/docs/orm/session.html#flushing).

**flush**:

When the Session is used with its default configuration, the flush step is nearly always done transparently. Specifically, **the flush occurs before any individual Query is issued, as well as within the commit() call before the transaction is committed**.

**commit**:

commit() is used to commit the current transaction. **It always issues flush() beforehand to flush any remaining state to the database**; this is independent of the “autoflush” setting. If no transaction is present, it raises an error. Note that the default behavior of the Session is that a “transaction” is always present; this behavior can be disabled by setting autocommit=True. In autocommit mode, a transaction can be initiated by calling the begin() method.

Ans2:

In the Hibernate Manual you can see this example

Session session = sessionFactory.openSession();

Transaction tx = session.beginTransaction();

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

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

session.save(customer);

if ( i % 20 == 0 ) { //20, same as the JDBC batch size

//flush a batch of inserts and release memory:

session.flush();

session.clear();

}

}

tx.commit();

session.close();

Without the call to the flush method, your first-level cache would throw an OutOfMemoryException

[Also you can look at this post about flushing](http://docs.jboss.org/hibernate/core/3.3/reference/en/html/objectstate.html#objectstate-flushing)

Q: Hibernate cannot simultaneously fetch multiple bags

Hibernate throws this exception during SessionFactory creation:

org.hibernate.loader.MultipleBagFetchException: cannot simultaneously fetch multiple bags

This is my test case:

**Parent.java**

@Entity

public Parent {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private Long id;

@OneToMany(mappedBy="parent", fetch=FetchType.EAGER)

// @IndexColumn(name="INDEX\_COL") if I had this the problem solve but I retrieve more children than I have, one child is null.

private List<Child> children;

}

**Child.java**

@Entity

public Child {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private Long id;

@ManyToOne

private Parent parent;

}

How about this problem? What can I do?

**EDIT**

OK, the problem I have is that another "parent" entity is inside my parent, my real behavior is this:

**Parent.java**

@Entity

public Parent {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private Long id;

@ManyToOne

private AntoherParent anotherParent;

@OneToMany(mappedBy="parent", fetch=FetchType.EAGER)

private List<Child> children;

}

**AnotherParent.java**

@Entity

public AntoherParent {

@Id

@GeneratedValue(strategy=GenerationType.IDENTITY)

private Long id;

@OneToMany(mappedBy="parent", fetch=FetchType.EAGER)

private List<AnotherChild> anotherChildren;

}

Hibernate doesn't like two collections with FetchType.EAGER, but this seems to be a bug, I'm not doing unusual things...

Removing FetchType.EAGER from Parent or AnotherParent solves the problem, but I need it, so real solution is to use @LazyCollection(LazyCollectionOption.FALSE) instead of FetchType (thanks to [Bozho](https://stackoverflow.com/users/203907/bozho) for the solution).

Ans:

I think a newer version of hibernate (supporting JPA 2.0) should handle this. But otherwise you can work it around by annotating the collection fields with:

@LazyCollection(LazyCollectionOption.FALSE)

Remember to remove the fetchType attribute from the @\*ToMany annotation.

But note that in most cases a Set<Child> is more appropriate than List<Child>, so unless you really need a List - go for Set

Hibernate:

Q: How to enable second level cache in Hibernate?

This is what you need to do:

1. Set the following Hibernate properties:
2. <property name="hibernate.cache.use\_second\_level\_cache">true</property>

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

1. Add an [ehcache.xml](http://ehcache.org/documentation/2.8/integrations/hibernate) file in your classpath, containing the cache configuration entries:
2. <cache name="com.mycompany.MyEntity"
3. maxElementsInMemory="50"
4. eternal="true"
5. overflowToDisk="false"
6. timeToIdleSeconds="600"
7. timeToLiveSeconds="600"
8. diskPersistent="false"
9. memoryStoreEvictionPolicy="LRU"

/>

1. Define the Caching type for each entity:
2. @Entity
3. @Cache(usage = CacheConcurrencyStrategy.READ\_WRITE)
4. public class MyEntity {
5. ...

}

Q: When and how to use hibernate second level cache?

Question:

I have trouble understanding when hibernate hits the second level cache and when does it invalidate the cache.

This is what I currently understand:

* Second level cache stores entities between sessions, scope is the SessionFactory
* You have to tell which entities to cache, no entity will get cached by default
* Query cache stores results of queries in the cache.

What I don't understand is

* When does hibernate hit this cache?
* Let's say I've set up the second level cache but not the query caching. I want to cache my customers, there's 50000 of them. In what ways can I retrieve the customers from the cache?
* I assume I can get them by id from cache. That would be easy but also not worthy of caching. But what if I want to do some calculation with all my customers. Let's say I want to show a list of the customers then how would I access them?
* How would I get all my customers if query caching is disabled?
* What would happen if someone updated one of the customers?
  + Would that customer get invalidated in the cache or would all customers get invalidated?

Or am I thinking caching totally wrong? What would be more appropriate uses of second level cache in that case? The hibernate documentation is not at all clear how the cache works in reality. There are only instructions on how to get it set up.

**Update:** So I've come to understand that second level cache(without query cache) would be good for loading data by id's. For example I have user object that I want to check for permissions in every request in a web application. Would this be a good case to reduce database access by caching the user in the second level cache? Like I would store the user id in the session or wherever and when I need to check for permissions I would load the user by it's id and check permissions.

Ans:

First of all, let's talk about process level cache (or 2nd level cache as they call it in Hibernate). To make it work, you should

1. configure cache provider
2. tell hibernate what entities to cache (right in hbm.xml file if you use this kind of mapping).

You tell to the cache provider how many objects it should store and when/why they should be invalidated. So let's say you have a Book and an Author entities, each time you're getting them from the DB, only those that are not in cache will be selected from actually DB. This increases performance significantly. It's useful when:

* You write to the database only via Hibernate (because it needs a way to know when to change or invalidate entities in the cache)
* You read objects often
* You have a single node, and you don't have replication. Otherwise you'll need to replicate the cache itself (use distributed caches like JGroups) which adds more complexity, and it doesn't scale as good as share-nothing apps.

So when does cache work?

* When you session.get() or session.load() the object that was previously selected and resides in cache. Cache is a storage where ID is the key and the properties are the values. So only when there is a possibility to search by ID you could eliminate hitting the DB.
* When your associations are lazy-loaded (or eager-loaded with selects instead of joins)

But it doesn't work when:

* If you don't select by ID. Again - 2nd level cache stores a map of entities' IDs to other properties (it doesn't actually store objects, but the data itself), so if your lookup looks like this: from Authors where name = :name, then you don't hit cache.
* When you use HQL (even if you use where id = ?).
* If in your mapping you set fetch="join", this means that to load associations joins will be used everywhere instead of separate select statements. Process level cache works on children objects only if fetch="select" is used.
* Even if you have fetch="select" but then in HQL you use joins to select associations - those joins will be issued right away and they will overwrite whatever you specified in hbm.xml or annotations.

Now, about Query Cache. You should note that it's not a separate cache, it's an addition to the process level cache. Let's say you have a Country entity. It's static, so you know that each time there will be the same result set when you say from Country. This is a perfect candidate for query cache, it will store a list of *IDs* in itself and when you next time select all countries, it will return this list to the process level cache and the latter, in turn, will return objects for each ID as these objects are stored already in the 2nd level cache. Query cache is invalidated each time anything related to the entity changes. So let's say you configured from Authors to be placed into a Query Cache. It won't be effective as Author changes often. So you should use Query Cache only for more or less static data.

Q: Hibernate get() vs load()

Ans: get() is eager method, it will fetch the records from the Db, where load is lazy, it will hit the DB on demand.

If object is not found in DB, get() returns null and load() will throw the ObjectNotFoundException.

load() will return proxy object.

Q: Hibernate hbm2ddl.auto possible values and what they do?

These are changes that could happen over DB:

* New tables
* new columns in old tables
* columns deleted
* data type of a column changed
* a type of a column changed it attributes
* tables have been dropped
* values of a column has changed

In each case what is the best solution?

From the [community documentation](https://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#configurations-hbmddl):

hibernate.hbm2ddl.auto Automatically validates or exports schema DDL to the database when the SessionFactory is created. With create-drop, the database schema will be dropped when the SessionFactory is closed explicitly.

e.g. validate | update | create | create-drop

So the list of possible options are,

* validate: validate the schema, makes no changes to the database.
* update: update the schema.
* create: creates the schema, destroying previous data.
* create-drop: drop the schema when the SessionFactory is closed explicitly, typically when the application is stopped.

These options seem intended to be developers tools and not to facilitate any production level databases, you may want to have a look at the following question; [Hibernate: hbm2ddl.auto=update in production?](https://stackoverflow.com/questions/221379/hibernate-hbm2ddl-autoupdate-in-production)

Q: JPA JoinColumn vs mappedBy?

What is the difference between:

@Entity

public class Company {

@OneToMany(cascade = CascadeType.ALL , fetch = FetchType.LAZY)

@JoinColumn(name = "companyIdRef", referencedColumnName = "companyId")

private List<Branch> branches;

...

}

and

@Entity

public class Company {

@OneToMany(cascade = CascadeType.ALL , fetch = FetchType.LAZY, mappedBy = "companyIdRef")

private List<Branch> branches;

...

}

Ans:

**@JoinColumn could be used on both sides of the relationship.** The question was about using @JoinColumn on the @OneToMany side (rare case). And the point here is in physical information duplication (column name) along with not optimized SQL query that will produce some additional UPDATE statements.

According to [documentation](http://docs.jboss.org/hibernate/annotations/3.5/reference/en/html_single/):

Since **many to one are** (almost) always the **owner side** of a bidirectional relationship in the JPA spec, the one to many association is annotated by @OneToMany(mappedBy=...)

@Entity

public class Troop {

@OneToMany(mappedBy="troop")

public Set<Soldier> getSoldiers() {

...

}

@Entity

public class Soldier {

@ManyToOne

@JoinColumn(name="troop\_fk")

public Troop getTroop() {

...

}

Troop has a bidirectional one to many relationship with Soldier through the troop property. You don't have to (must not) define any physical mapping in the mappedBy side.

To map a bidirectional one to many, with the **one-to-many side as the owning side**, you have to remove the mappedBy element and set the many to one @JoinColumn as insertable and updatable to false. This solution is not optimized and will produce some additional UPDATE statements.

@Entity

public class Troop {

@OneToMany

@JoinColumn(name="troop\_fk") //we need to duplicate the physical information

public Set<Soldier> getSoldiers() {

...

}

@Entity

public class Soldier {

@ManyToOne

@JoinColumn(name="troop\_fk", insertable=false, updatable=false)

public Troop getTroop() {

...

}

Ans2:

The annotation @JoinColumn indicates that this entity is the owner of the relationship (that is: the corresponding table has a column with a foreign key to the referenced table), whereas the attribute mappedBy indicates that the entity in this side is the inverse of the relationship, and the owner resides in the "other" entity. This also means that you can access the other table from the class which you've annotated with "mappedBy" (fully bidirectional relationship).

In particular, for the code in the question the correct annotations would look like this:

@Entity

public class Company {

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

private List<Branch> branches;

}

@Entity

public class Branch {

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "companyId")

private Company company;

}

The [JPA 2.0 Specification](http://www.jcp.org/en/jsr/detail?id=317) states that:

* The entity class must have a no-arg constructor. It may have other constructors as well. The no-arg constructor must be public or protected.
* The entity class must a be top-level class. An enum or interface must not be designated as an entity.
* The entity class must not be final. No methods or persistent instance variables of the entity class may be final.
* **If an entity instance is to be passed by value as a detached object** (e.g., through a remote interface), the entity class must implement the Serializable interface.
* Both abstract and concrete classes can be entities. Entities may extend non-entity classes as well as entity classes, and non-entity classes may extend entity classes.

The specification contains no requirements about the implementation of equals and hashCode methods for entities, only for primary key classes and map keys as far as I know.