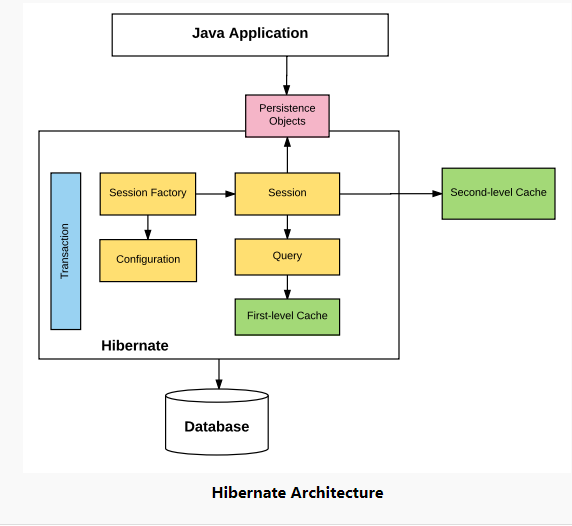
Hibernate is an open source **Java persistence** framework project. It performs powerful object-relational mapping and query databases using HQL and SQL. Hibernate is a great tool for **ORM mappings** in Java.



Hibernate is an object-relational mapping (ORM) library for the Java language, providing a framework for mapping an object-oriented domain model to a traditional relational database. This means you are not required to build and execute SQL queries for interaction with database.

1. **Configuration:** Generally written in:

* hibernate.properties
* hibernate.cfg.xml (This configuration file will be used to store database connection information and schema level settings.)

For Java configuration, we can find class annotated with @Configuration. It is used by Session Factory to work with Java Application and the Database. It represents an entire set of mappings of an application Java Types to an SQL database.

1. **Session Factory**: Any user application requests Session Factory for a session object. Session Factory uses configuration information from above listed files, to instantiates the session object appropriately.

Session factory is used to obtain the connection of database and various activities like commit and rollback.

1. **Session:** This represents the interaction between the application and the database at any point of time. This is represented by the org.hibernate.Session class. The instance of a session can be retrieved from the **SessionFactory** bean.
2. **Query:** It allows applications to query the database for one or more stored objects. Hibernate provides different techniques to query database, including **NamedQuery** and **Criteria API**.
3. **First-level cache:** It represents the default cache used by Hibernate Session object while interacting with the database. It is also called as session cache and caches objects within the current session. All requests from the Session object to the database must pass through the first-level cache or session cache. One must note that the first-level cache is available with the session object until the Session object is live.
4. **Transaction**: Enables us to achieve data consistency, and rollback in case something goes unexpected.

**Features of the Hibernate framework**

#### **Object/Relational Mapping:** Hibernate, as an ORM framework, allows the mapping of the Java domain object with database tables and vice versa. As a result, business logic can access and manipulate database entities via Java objects.

#### **JPA provider:**

Hibernate does support the [Java Persistence API](https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html)**(JPA)** specification. JPA is a set of specifications for accessing, persisting, and managing data between Java objects and relational database entities.

#### **Idiomatic persistence:**

Any class that follows [object-oriented principles](https://howtodoinjava.com/oops/object-oriented-principles/) such as inheritance, polymorphism, and so on, can be used as a persistent class.

#### **High performance and scalability:**

Hibernate supports techniques such as different fetching strategies, lazy initialization, optimistic locking, and so on, to achieve high performance, and it scales well in any environment.

#### **Easy to maintain:**

Hibernate is easier to maintain as it requires no special database tables or fields. It generates SQL at system initialization time. It is much quicker and easier to maintain compared to JDBC.

**Relation of hibernate with JPA**

[JPA (Java Persistence API)](https://en.wikipedia.org/wiki/Java_Persistence_API) is a specification for persistence providers to implement. Hibernate is one such implementation of JPA specification. We can annotate our classes as much as we would like with JPA annotations, however without an implementation nothing will happen

When we use hibernate with JPA we are using the Hibernate JPA implementation. The benefit of this is that we can swap out hibernates implementation of JPA for another implementation of the JPA specification.

**Hibernate load entity – session.load ()**

Hibernate’s Session interface provides several overloaded load () methods for loading entities from the database. Each load () method requires the object’s **primary key** as an identifier, and it is mandatory to provide it.

**Difference between load () and get () methods**

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

**Hibernate entity class**

Hibernate talks with Java POJO classes which are marked as hibernate entities. To convert a Java class into entity, we need to apply **@Entity** annotation on it.

There are other hibernate annotations like:

* @Table
* @Column
* @Id

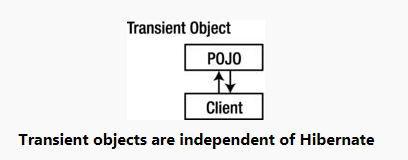
These mappings help in mapping entity fields to database table and columns.

# **Hibernate Entity / Persistence Lifecycle States**

1. **Transient Object:**

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

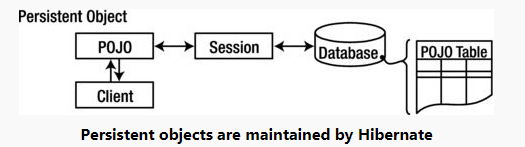
To persist the changes to a transient object, we would have to ask the session to save the transient object to the database, at which point Hibernate assigns the object an identifier and marks the object as being in persistent state.



1. **Persistent Object:**

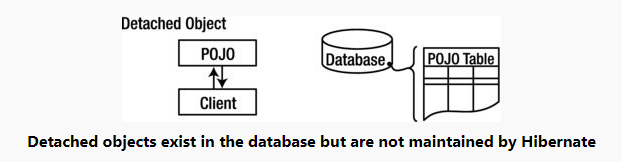
Persistent objects exist in the database and Hibernate manages the persistence for persistent objects.

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.



1. **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.



To persist changes made to a detached object, the application must reattach it to a valid Hibernate session

1. **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.

**Bullet Points**

1. 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.
2. Persistent object represents 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.
3. 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.
4. 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 save () and saveOrUpdate () methods**

We should not call save () method on a persistent entity (entity associated with any hibernate session). Any changes done to persistent entity is automatically saved.



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.

**Bullet Points:**

1. 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.
2. SaveOrUpdate () calls either save () or update () based on identifier exists or not. e.g. if identifier does not exist, save () will be called or else update () will be called.
3. Probably you will get very few chances to call save () or saveOrUpdate () methods, as hibernate manages all changes done in persistent objects.

**Refreshing Hibernate Entities Using refresh () Method**

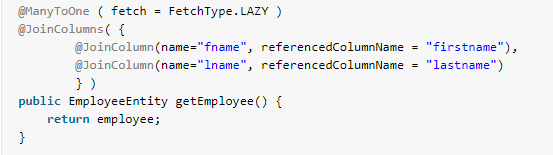
Sometimes we face situation where we application database is modified with some external application/agent and thus corresponding hibernate entity in your application becomes out of sync with its database representation i.e. having old data. In this case, you can use **session. Refresh () method to re-populate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

**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.

[**Lazy loading**](https://en.wikipedia.org/wiki/Lazy_loading) is a design pattern commonly used 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.

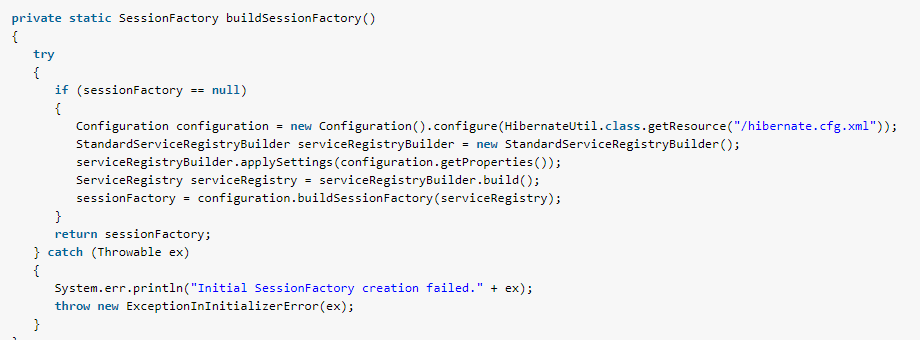


**AnnotationConfiguration:**

**Used to create SessionFactory in Hibernate 3**

**Classes used in building SessionFactory in hibernate 4.**

1. **Configuration:** In place of deprecated AnnotationConfiguration
2. **StandardServiceRegistryBuilder:** In place of deprecated ServiceRegistryBuilder



**Bullet Points**

1. Requesting a persistent object again from the same Hibernate session returns the “same java instance” of a class.
2. Requesting a persistent object from the different Hibernate session returns “different java instance” of a class.
3. As a best practice, we should always implement [equals() and hashCode ()](https://howtodoinjava.com/java/related-concepts/working-with-hashcode-and-equals-methods-in-java/) methods in our hibernate entities; and always compare them using equals()method only.

**JPA Cascade Types**

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

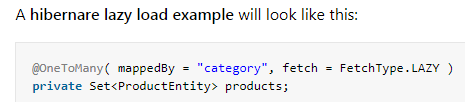
1. **CascadeType.PERSIST**: cascade type persist means that save () or persist () operations cascade to related entities.
2. **CascadeType.MERGE**: cascade type merge means that related entities are merged when the owning entity is merged.
3. **CascadeType.REFRESH**: cascade type refresh does the same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

Hibernate fetches data from database either in eager or lazy mode.

**Hibernate lazy loading** refer to strategy when data is loaded lazily, on demand.

**Eager loading** will load association entity as well when owner entity is fetched first time.

[@OneToMany](https://howtodoinjava.com/hibernate/hibernate-one-to-many-mapping-using-annotations/) and @ManyToMany associations are defaulted to **LAZY** loading; [@OneToOne](https://howtodoinjava.com/hibernate/hibernate-one-to-one-mapping-using-annotations/) and @ManyToOne are defaulted to **EAGER** loading.



[Hibernate](https://howtodoinjava.com/hibernate-tutorials/) provides 3 different ways to retrieve data from database.

1. [HQL](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/" \o "Complete Hibernate Query Language [HQL] Tutorial" \t "_blank)
2. [native SQL queries](https://howtodoinjava.com/hibernate/complete-hibernate-query-language-hql-tutorial/" \o "Complete Hibernate Query Language [HQL] Tutorial" \t "_blank).
3. **Hibernate criteria queries**.

**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.

Ex:

Criteria crit = session.createCriteria(Product.class);

List<Product> results = crit.list();

#### **Restrictions.eq() Example**



#### **Restrictions.ne() Example**



#### **HQL – Hibernate Query Language**

HQL is an object-oriented query language, like SQL, but instead of operating on tables and columns.

HQL works with persistent objects and their properties. This is main difference between **hql vs sql**.

HQL is a superset of the JPQL, the Java Persistence Query Language. A JPQL query is a valid HQL query, but not all HQL queries are valid JPQL queries

HQL syntax is defined as an [ANTLR](https://en.wikipedia.org/wiki/ANTLR) grammar. The grammar files are included in the grammar directory of the Hibernate core download. (*ANTLR is a tool for building language parsers*).

#### **HQL Update Statement**

#### **HQL Delete Statement**



#### **HQL Insert Statement**

An HQL INSERT cannot be used to directly insert arbitrary entities—it can only be used to insert entities constructed from information obtained from SELECT queries (unlike ordinary SQL, in which an INSERT command can be used to insert arbitrary data into a table, as well as insert values selected from other tables).

#### **HQL Logs**

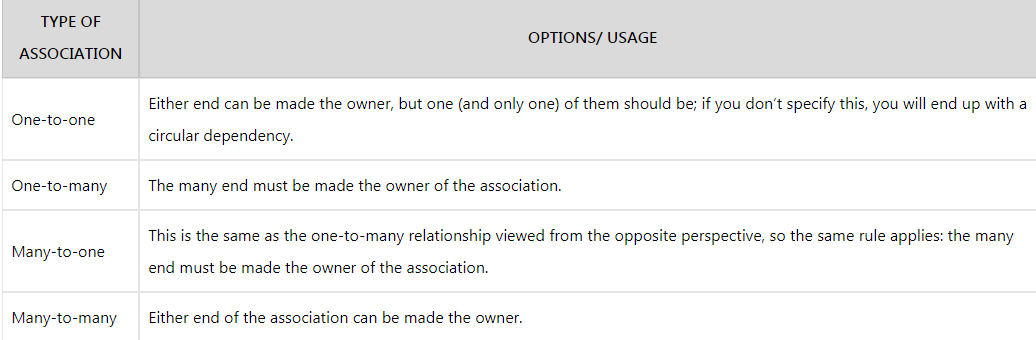
The easiest way to see the SQL for a Hibernate HQL query is to enable SQL output in the logs with the “*show\_sql*” property. Set this property to true in your **hibernate.cfg.xml** configuration file and Hibernate will output the SQL into the logs.

# **Hibernate @NamedQuery**

Named queries in hibernate is a**technique to group the HQL statements in single location.**

**Advantage:**

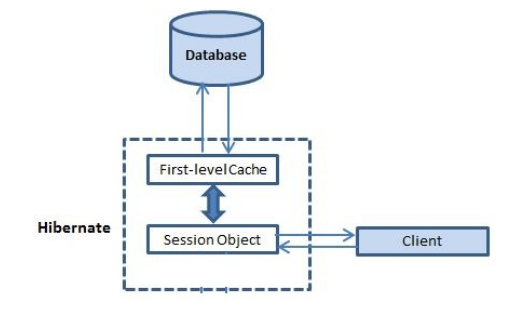
1. Fail fast: Their syntax is checked when the session factory is created, making the application fail fast in case of an error.
2. Reusable: They can be accessed and used from several places which increase re-usability.



# **Hibernate First Level Cache**

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

1. **Fist level cache** in hibernate is enabled by default.
2. First level cache is associated with **Session object.**
3. 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**

1. First level cache is associated with “session” object and other session objects in application cannot see it.
2. The scope of cache objects is of **session**. Once session is closed, cached objects are gone forever.
3. First level cache is **enabled** **by** **default** and you cannot disable it.
4. When we query an entity first time, it is retrieved from database and stored in first level cache associated with hibernate session.
5. If we query same object again with same session object, it will be loaded from cache and no sql query will be executed.
6. 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.
7. The whole session cache can be removed using **clear ()** method. It will remove all the entities stored in cache.

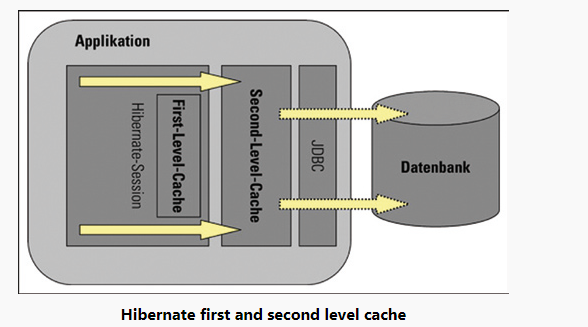
# **Hibernate Second Level Cache**

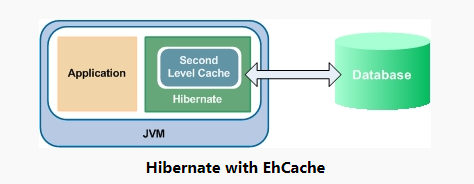
**Fist level cache**: This is enabled by default and works in session scope.

**Second level cache**: This is apart from first level cache which is available to be used globally in session factory scope.

**second level cache is created in session factory scope** and is **available to be used in all sessions** which are created using that session factory.

It also means that **once session factory is closed, all cache associated with it die** and cache manager also closed.





**How second level cache works**

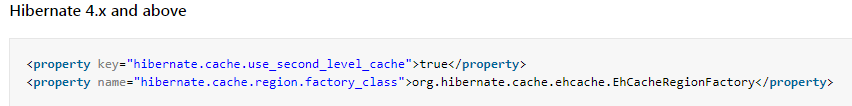
1. Whenever hibernate session try to load an entity, the very first place it looks for cached copy of entity in first level cache (associated with hibernate session).
2. If cached copy of entity is present in first level cache, it is returned as result of load method.
3. If there is no cached entity in first level cache, then second level cache is looked up for cached entity.
4. 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 needs to go to second level cache again.
5. 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.
6. Second level cache validate itself for modified entities if modification has been done through hibernate session APIs.
7. 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.

**Terracotta Ehcache is a popular open source Java cache that can be used as a Hibernate second level cache**

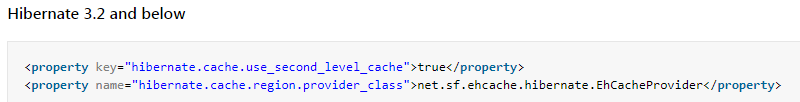
**Configuring EhCache**

To configure ehcache, you need to do two steps:

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





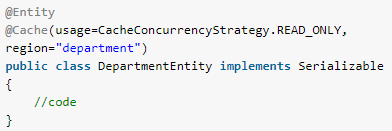


**Configuring entity objects**

1. If you are using **hbm.xml files** then use below configuration:



1. Otherwise, if you are using annotations, use these **annotations**:



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

* **none**: No caching will happen.
* **read-only**: If your application needs to read, but not modify, instances of a persistent class, a read-only cache can be used.
* **read-write**: If the application needs to update data, a read-write cache might be appropriate.
* **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.
* **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.

# **Hibernate OSCache Configuration**

[**OSCache**](https://howtodoinjava.com/category/frameworks/oscache/) is a Java framework developed by OpenSymphony that makes it easy to cache content in Web applications. With hibernate, it can be configured to act as [**second level cache**](https://howtodoinjava.com/hibernate/how-hibernate-second-level-cache-works/).

## ****Hibernate configuration****

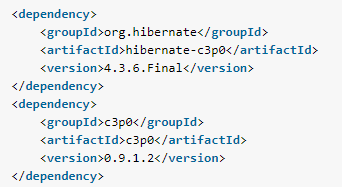


**Hibernate C3P0 Connection Pool Configuration**

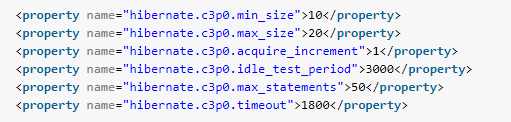
1. By default, hibernate uses JDBC connections to interact with a database.
2. Creating these connections is expensive—probably the most expensive single operation Hibernate will execute in a typical-use case.
3. Since JDBC connection management is so expensive that possibly you will advise to use a pool of connections, which can open connections ahead of time (and close them only when needed, as opposed to “when they’re no longer used”).

[**C3P0**](http://www.mchange.com/projects/c3p0/)  is an example of an external connection pool

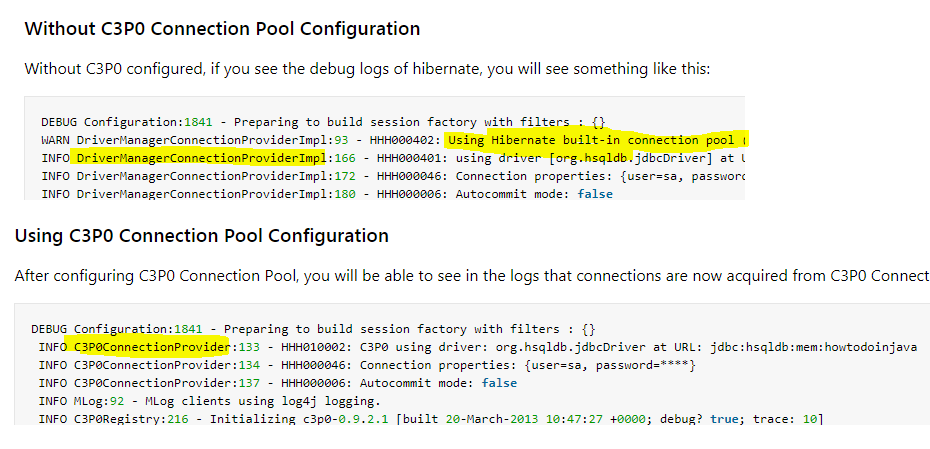
**Maven dependencies**



**Configure C3P0 Connection Pool with Hibernate**



**Test connection pooling in runtime:**



# Hibernate / JPA 2 Persistence Annotations

