OBJECT™
TECHNOLOGIES

Advanced Hibernate

What Will Be Covered

OBJECT

- Entity states
- Hibernate Caching
- Transaction management

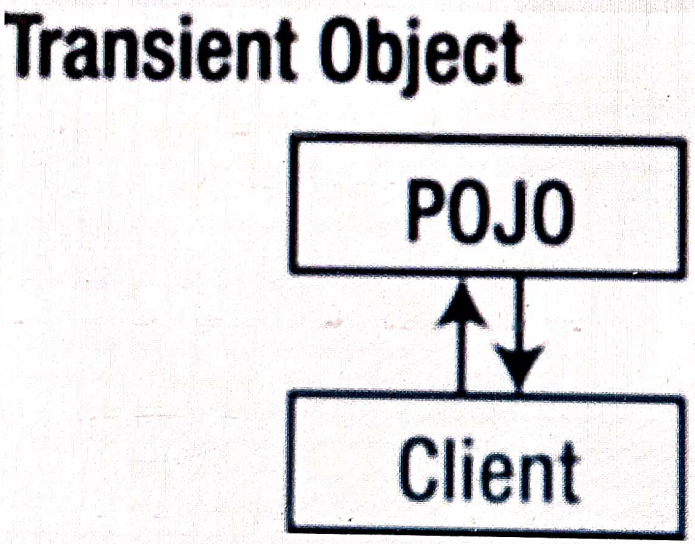
Hibernate Persistent States

OBJECT

- **Hibernate** works with normal Java objects that your application creates with the new operator.
- In raw form (without annotations/mapping), hibernate will not be able to identify your java classes; but when they are properly annotated with required annotations or when mapping information written in XML file then hibernate will be able to identify them and then work with them e.g. store in DB, update them etc.
- These objects can be said to mapped with hibernate.
- 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**.

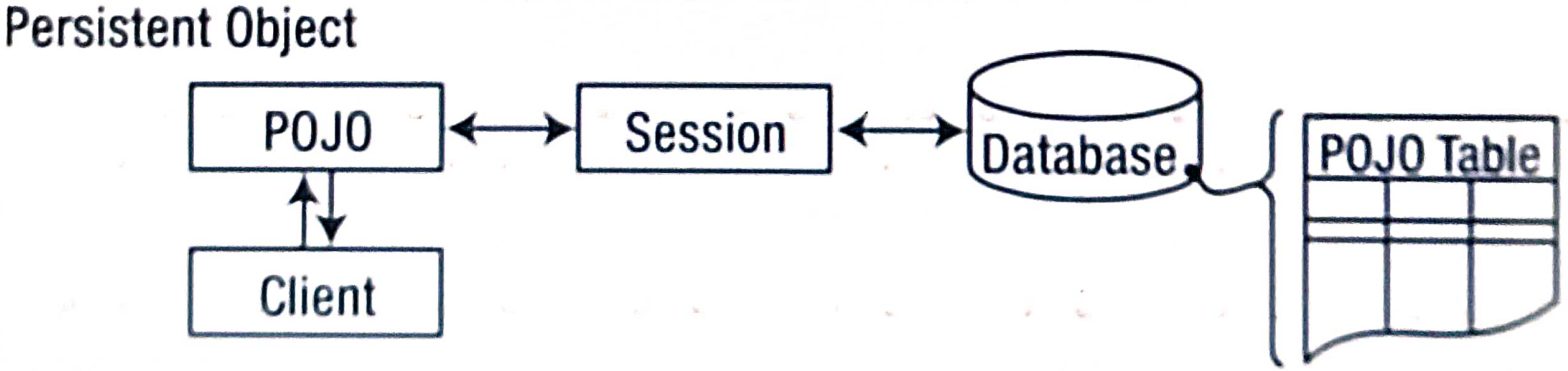
Transient Object

- Transient objects exist in heap memory. Hibernate does not manage transient objects or persist changes to transient objects.
- When the object is created newly, it is said to be in transient state
- Transient objects are independent of Hibernate



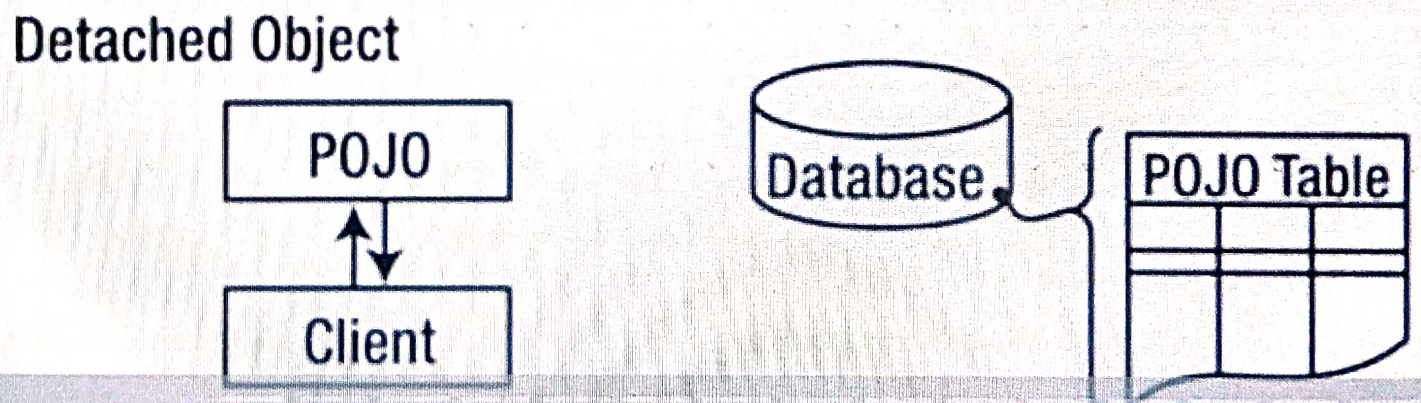
Persistent Object

- 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.
- Persistent objects are maintained by Hibernate
- When the object is in persistent state, then it represent one row of the database, if the object is in persistent state then it is associated with the unique Session



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



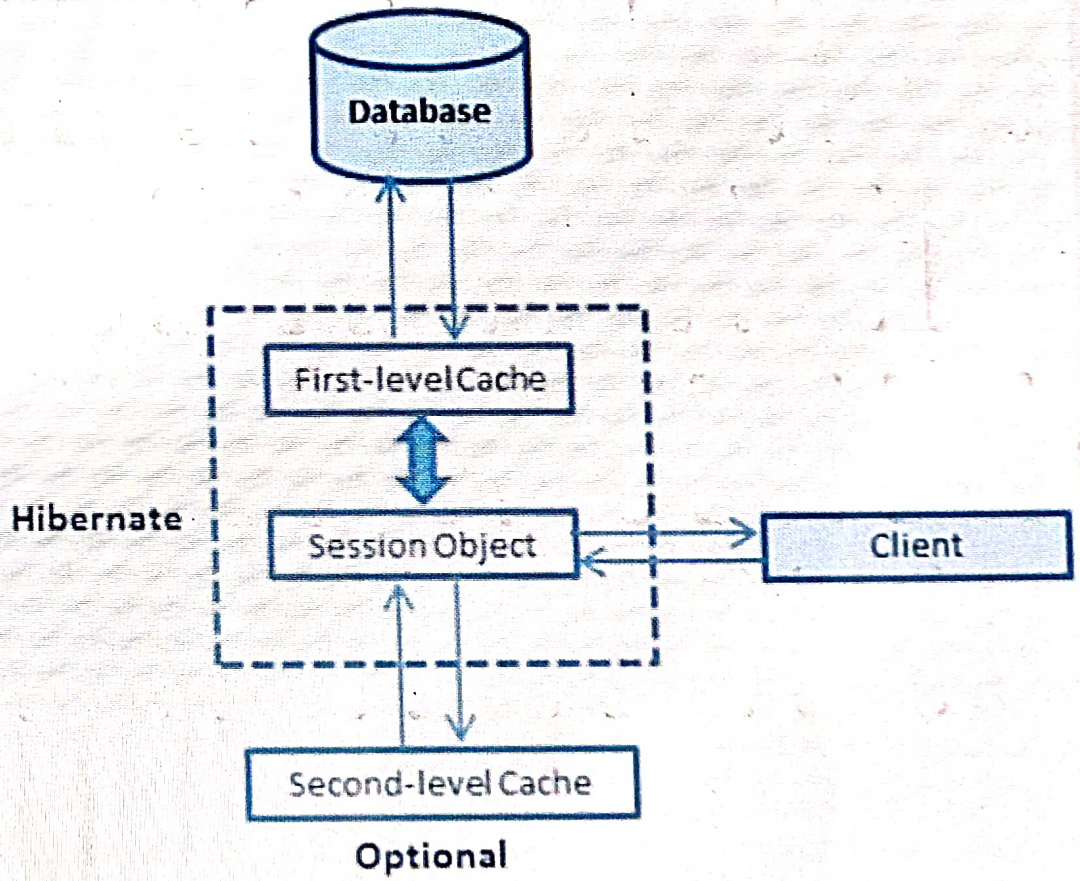
Removed Object

- 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.

Hibernate Caching

OBJECT

- Caching is all about application performance optimization and it sits between your application and the database to avoid the number of database hits as many as possible to give a better performance for performance critical applications.
- There are mainly two types of caching: first level cache and second level cache.



First Level Cache

OBJECT

- The first-level cache is the Session cache and is a mandatory cache through which all requests must pass. The Session object keeps an object under its own power before committing it to the database.
- If you issue multiple updates to an object, Hibernate tries to delay doing the update as long as possible to reduce the number of update SQL statements issued. If you close the session, all the objects being cached are lost and either persisted or updated in the database.
- Session object holds the first level cache data. It is enabled by default. The first level cache data will not be available to entire application. An application can use many session object.

Second Level Cache

OBJECT

- **SessionFactory object holds the second level cache data. The data stored in the second level cache will be available to entire application. But we need to enable it explicitely.**
- Second level cache is an optional cache and first-level cache will always be consulted before any attempt is made to locate an object in the second-level cache. The second-level cache can be configured on a per-class and per-collection basis and mainly responsible for caching objects across sessions.
- **Hibernate second level cache** uses *a common cache for all the session object of a session factory*. It is useful if you have multiple session objects from a session factory.
- **SessionFactory** holds the second level cache data. It is global for all the session objects and not enabled by default.

Second Level Cache

OBJECT

- The Hibernate second-level cache is set up in two steps. First, you have to decide which concurrency strategy to use. After that, you configure cache expiration and physical cache attributes using the cache provider.
- Any third-party cache can be used with Hibernate.
  An **org.hibernate.cache.CacheProvider** interface is provided, which must be implemented to provide Hibernate with a handle to the cache implementation.
- 3 extra steps for second level cache example using EH cache
- Add 2 configuration setting in hibernate.cfg.xml file

```
<property name="cache.provider_class">org.hibernate.cache.EhCacheProvider</property>
<property name="hibernate.cache.use_second_level_cache">true</property>
```

- Add cache usage setting in hbm file

Second Level Cache

OBJECT

```
<cache usage="read-only" />
```

- Create ehcache.xml file

```
<?xml version="1.0"?>
<ehcache>

<defaultCache
maxElementsInMemory="100"
eternal="true"/>

</ehcache>
```

Concurrency strategies

OBJECT

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

- **Transactional:** Use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions,in the rare case of an update.
- **Read-write:** Again use this strategy for read-mostly data where it is critical to prevent stale data in concurrent transactions,in the rare case of an update.
- **Nonstrict-read-write:** This strategy makes no guarantee of consistency between the cache and the database. Use this strategy if data hardly ever changes.
- **Read-only:** A concurrency strategy suitable for data which never changes. Use it for reference data only.

Cache Providers

OBJECT

- Different vendors have provided the implementation of Second Level Cache.
- Each implementation provides different cache usage functionality.
- Few exmples of cache providers :
  - EH Cache
  - OS Cache
  - Swarm Cache
  - JBoss Cache

Transaction management

OBJECT

- A **transaction** simply represents a unit of work. In such case, if one step fails, the whole transaction fails (which is termed as atom
- In hibernate framework, we have **Transaction** interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA,JDBC).icity). A transaction can be described by ACID properties
- A transaction is associated with Session and instantiated by calling **session.beginTransaction()**.
- Few functionalities that are available
  - **void begin()** starts a new transaction.
  - **void commit()** ends the unit of work unless we are in FlushMode.NEVER.
  - **void rollback()** forces this transaction to rollback.
  - **boolean isAlive()** checks if the transaction is still alive.

Advanced hibernate

In this chapter we will throw light on some advanced concepts like hibernate caching, entitiy states etc.

Entity states

Given an instance of a class that is managed by persistent context, it can be in any one of four different persistence states (known as hibernate entity lifecycle states):

Transient

Transient entities exist in heap memory as normal Java objects. Hibernate does not manage transient entities. The persistent context does not track the changes done on them. In simple words, a transient entity has neither any representation in the datastore nor in the current Session. A transient entity is simply a POJO without any identifier.

```
EmployeeEntity employee = new EmployeeEntity();
```

Persistent or Managed

Persistent entities exist in the database, and Hibernate's persistent context tracks all the changes done on the persistent entities by the client code. A persistent entity is mapped to a specific database row, identified by the ID field. Hibernate's current running Session is responsible for tracking all changes done to a managed entity and propagating these changes to database.

We can get persistent entity in either of two ways:

- Load the entity using get() or load() method.
- Persist the transient or detached entity using persist(), save(), update() or saveOrUpdate() methods.

```
EmployeeEntity employee = session.load(1);
//or
EmployeeEntity employee = new EmployeeEntity();
session.save(employee);
```

Detached

Detached entities have a representation in the database but these are currently not managed by the Session. Any changes to a detached entity will not be reflected in the database, and vice-versa.

A detached entity 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.

Detaching an entity from Session

Detached

Detached entities have a representation in the database but these are currently not managed by the Session. Any changes to a detached entity will not be reflected in the database, and vice-versa.

A detached entity 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.

Detaching an entity from Session

```
session.close();
//or
session.evict(entity);
```

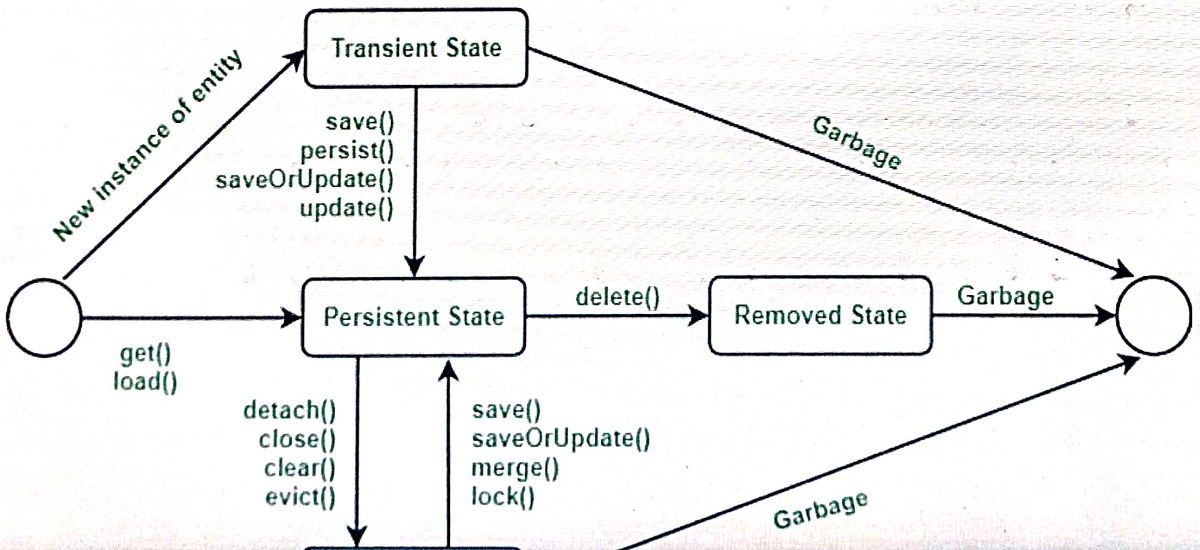
A detached instance can be associated with a new Hibernate session when your application calls one of the load(), refresh(), merge(), update(), or save() methods on the new session with a reference to the detached object.

Removed

Removed entities are objects that were being managed by Hibernate (persistent entities, in other words) and now those 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 entities are deleted.

```
session.remove(employee);
```

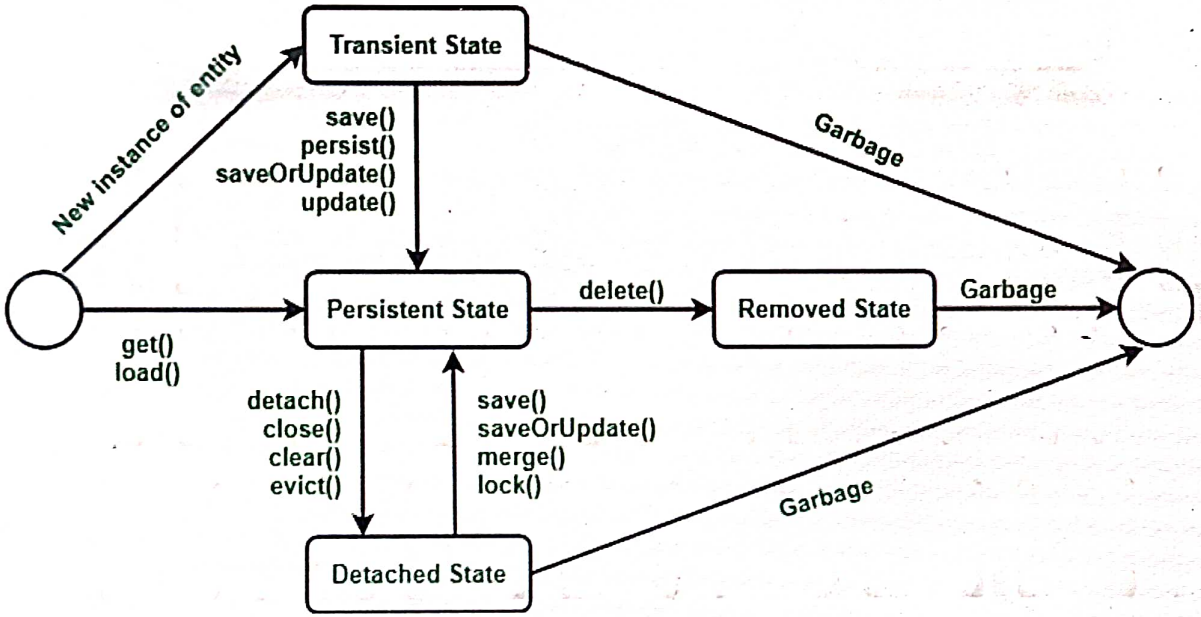


Removed

Removed entities are objects that were being managed by Hibernate (persistent entities, in other words) and now those 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 entities are deleted.

```
session.remove(employee);
```



Transaction management

A transaction simply represents a unit of work. Generally speaking, a transaction is a set of SQL operations that need to be either executed successfully or not at all.

In the hibernate framework, we have a Transaction interface that defines the unit of work. It maintains abstraction from the transaction implementation (JTA, JDBC).

A transaction is associated with Session and instantiated by calling session.beginTransaction().

The methods of the Transaction interface

void begin() - starts a new transaction.

void commit() - ends the unit of work unless we are in FlushMode.NEVER.

void rollback() - forces this transaction to rollback.

The methods of the Transaction interface

void begin() - starts a new transaction.

void commit() - ends the unit of work unless we are in FlushMode.NEVER.

void rollback() - forces this transaction to rollback.

void setTimeout(int seconds) - it sets a transaction timeout for any transaction started by a subsequent call to begin on this instance.

boolean isAlive() - checks if the transaction is still alive.

Example

```
Transaction transaction = null;
try (Session session = HibernateUtil.getSessionFactory().openSession()) {
        // start a transaction
        transaction = session.beginTransaction();
        // Delete a student object
        Student student = session.get(Student.class, id);
        if (student != null) {
             String hql = "DELETE FROM Student " + "WHERE id = :studentId";
             Query query = session.createQuery(hql);
             query.setParameter("studentId", id);
             int result = query.executeUpdate();
             System.out.println("Rows affected: " + result);
        }
        // commit transaction
        transaction.commit();
  } catch (Exception e) {
        if (transaction != null) {
           transaction.rollback();
        }
  }
```

Assignments

1. Try to load one instance twice within a single session. Verify whether query gets executed once or twice
2. Try to modify the entity from persistent state and detached state and check whether those changes are reflected in the database